OpenADK (Agent Development Kit) for Java

Developer's Guide

Version 2.4

PEARSON

Data Solutions
9815 S. Monroe St.,
Ste. 400
Sandy, UT 84070
1.877.790.1261

www.pearsondatasolutions.-
com

Contents

Part I

1. Introduction

The OpenADK for Java – or ADK – is a high-level class framework for the Java platform that simplifies the job of writing sophisticated, feature-rich SIF Agents for the Schools Interoperability Framework™. It was designed to insulate applications from the low-level mechanics of the SIF messaging infrastructure and XML data model. With the ADK developers focus on the integration logic of their applications instead of the underlying details of the SIF protocol.

Some of the ADK's features include:

- High level, object-oriented class framework
- Supports all versions of SIF, including 2.1, 2.0r1, 2.0, 1.5r1, and 1.1
- Connects to any Zone Integration Server that supports the SIF standard
- SIF Data Object libraries to model all SIF objects as object-oriented Java classes
- Handles message parsing and rendering for all versions of the SIF data model
- Features a powerful data mapping engine to transform data to and from SIF XML
- HTTPS and HTTP support (agents include an embedded web server)
- Supports both Push and Pull modes of operation
- Supports simultaneous connectivity with multiple zones
- Supports zone aggregation via "topic" classes
- Support for synchronous queries and Selective Message Blocking (SMB)

SIF Versions Supported

The ADK supports all published versions of the Schools Interoperability Framework specification 1.1 and beyond.

Specification	Date
SIF 2.4	June 11, 2010
SIF 2.3	February 12, 2009
SIF 2.2	March 17, 2008
SIF 2.1	September 18, 2007
SIF 2.0r1	June 19, 2007
SIF 2.0	October 18, 2006
SIF 1.5r1	October 11, 2004
SIF 1.1	February 6, 2003

Handling of Multiple SIF Versions

The ADK supports multiple versions of the Schools Interoperability Framework concurrently. The framework sends and receives messages and renders SIF Data Objects according to the unique rules of each version of the SIF Specification. Details such as XML element names and sequence are automatically taken into account when rendering and parsing data objects.

When an agent initializes the class framework by calling the `ADK.initialize()` method at startup, it may specify the version of the SIF Specification the ADK uses to register with zone integration servers. This version is global to the Java virtual machine; if not specified, the most recently published version of the spec is used. The SIF Version passed to the `ADK.initialize` method is considered to be the agent's default version, which will be used to render all messages that originate from the agent—for example, when a SIF_Event message is reported to a zone.

Messages received by the agent from the zone integration server are automatically processed using the version of SIF to which the message conforms. For example, if your agent registers in a zone as a SIF 1.1 agent it can still receive and process older SIF 1.1 messages as well as newer 2.0r1 messages. The ADK always takes care of parsing incoming messages and rendering outbound messages in the appropriate version of SIF.

Requirements

Java Requirements

- Java 2 Standard Edition SDK 1.5.0 or later (for development)
- Java 2 Runtime Edition 1.5.0 or later (for deployment)
- A Java IDE such as Eclipse is recommended for development but not required. Eclipse can be downloaded free of charge from www.eclipse.org
- Apache Ant is required to run the Ant scripts to rebuild the ADK Example agents from source code. Ant can be downloaded free of charge from www.apache.org

SIF Requirements

Agents built with the ADK connect to all zone integration server (ZIS) products that conform to the SIF Specification 1.0r1 and later.

License & Redistribution

The ADK is licensed under the Apache 2.0 Software License. Copyright in the ADK is held by Pearson and by the community contributors to the ADK project. The license is in the file LICENSE-2.0.txt in the ADK folder after installation. Note that the Apache license does not require you to make any of your application's code available as open source software.

All files under the Apache license may be redistributed in binary and/or source form with your agent, although only a subset is actually needed by most agents (the contents of the `lib` directory are usually required). Note that some files may be under the same or a different open source license and subject to third-party license agreements included in the `licenses` directory.

Documentation

Javadoc

The Javadoc is an integral part of the ADK documentation. To view it, open the `docs/Javadoc/index.html` file in a browser of your choice.

Developer's Guide

This guide is divided into three parts:

▪ Introduction & Installation	*Page 6*
▪ Concepts	*Page 10*
▪ Core Classes	*Page 43*

If you are an experienced Java developer and have a good understanding of the Schools Interoperability Framework, you may want to skim through the Developer's Guide first and then experiment with the ADK Example agents. These programs demonstrate the basic structure of agents developed with the ADK. Once you are familiar with the core classes—ADK, Agent, Topic, Zone, Publisher, Subscriber, and QueryResults—you should have little trouble building and testing your own SIF Agent.

For those not familiar with SIF, we strongly suggest consulting the latest SIF Specification available from the sifinfo.org web site. The ADK Developer's Guide assumes a basic understanding of the SIF architecture and concepts.

Tech Notes

Data Solutions may post additional supplemental documentation regarding the ADK and SIF Agent development to the Internet. These documents are in the form of *Tech Notes* or articles.

Technical Support

Support for the ADK is provided through the community web site tools.

2. Installation

Installing the ADK

To install the ADK, run the installation program and follow the instructions presented in the dialog boxes. Installers are available for Microsoft Windows, Mac OS X, Linux, and Solaris. You can also download the .zip package and expand it to any folder on your local file system. The ADK is self-contained and does not make any changes to the Windows Registry or to external files on other operating systems.

Directory Contents

The ADK directory structure:

`Root`	Licenses and Readme files

`\docs`	PDF Documentation and Javadocs
`\examples`	ADK Example Agents
`\lib`	Java libraries files that must be on the classpath to compile or run agents developed with the ADK. You may redistribute any of the libraries in this directory. Some files, such as the Xerces XML parser from Apache Software Foundation, may only be redistributed in accordance with open-source license restrictions.

Java Classpath

Once installed, make sure the JAR files from the `lib` directory are referenced by your development environment's Java classpath.

Note the `lib` directory includes a number of JAR files beginning with the prefix "sdo". These are individual SIF Data Object modules. The SIF Data Objects library is divided into multiple JARs so that you can redistribute only those components used by your agent to reduce your agent's memory and disk footprint. The `sdoall.jar` file includes all SIF Data Object modules packaged into a single JAR, which can be redistributed in place of the individual library files.

Part II

ESSENTIAL CONCEPTS

3. Concepts

The Role of the ADK Class Framework

The ADK is best characterized as a high-level Java-class framework, as opposed to a set of programming APIs. As a class framework, it places certain roles and responsibilities on your application and others on the framework itself. In general, the ADK takes care of all of the details of the SIF infrastructure, messaging, and data encapsulation while your agent handles the business logic and data manipulation that is unique to your application.

Note the ADK does not contain any database or user interface classes as these are outside the scope of a SIF class framework.

Agent Responsibilities

When writing an agent with the ADK, you're typically responsible for the following:

- **Implement an Agent Class**

  Your agent should derive a class from the ADK's abstract `Agent` class. The Agent class is usually where the main program entry point is found and where startup and shutdown tasks are centralized.

- **Implement Message Handlers**

  The ADK interprets and dispatches incoming SIF messages to one or more *message handler* classes implemented by your agent. A message handler is nothing more than a Java class that implements the *Subscriber, Publisher*, or *QueryResults* interfaces (and *ReportPublisher* for Vertical Reporting agents). Most of these interfaces have one or two methods, so it is easy to think of message handlers as callback functions.

  Because message handlers are interface-based, you have a tremendous amount of flexibility in the way you organize your code. For example, you could design message handler classes so that a single class handles SIF messaging for a variety of object types and from many zones. Or, you could create individual message handlers to encapsulate each object type supported by your agent, each zone your agent is connected to, or a combination of both. Regardless of how you choose to organize your code, the ADK will call your message handler when it receives an incoming SIF_Event, SIF_Request, or SIF_Response message from the zone integration server.

  - Implement one or more *Subscriber* message handlers if your agent will respond to `SIF_Event` messages. When connecting to SIF Zones, the ADK automatically sends a SIF_Subscribe message for each SIF Data Object for which a *Subscriber* message handler exists.

  - Implement one or more *Publisher* message handlers if your agent will respond to SIF_Request messages. When connecting to SIF Zones, the ADK automatically sends a SIF_Provide message for each SIF Data Object for which a *Publisher* message handler exists.

  - Implement one or more *QueryResults* message handlers if your agent sends SIF_Request messages for data and will process the resulting SIF_Response query results.

  - Implement one or more *ReportPublisher* message handlers if your agent will respond to SIF_Request messages for Vertical Reporting objects.

- **Connect to SIF Zones**

  Most agents are designed to connect to one or more SIF Zones at startup, often by reading a list of zone connection parameters from a configuration file. (Tip: You can use the AgentConfig class in the `openadk.library.tools.cfg` package to read from an XML-based configuration file.) Agents developed with the ADK can connect to multiple zones concurrently. When connecting to a zone, the class framework takes care of sending the appropriate SIF provisioning messages (SIF_Register, SIF_Subscribe, and SIF_Provide) to the server based on the message handlers you have implemented. For instance, if you've implemented a *Subscriber* message handler to respond to StudentPersonal SIF_Events, the ADK will automatically send a SIF_Subscribe for StudentPersonal each time it connects to a zone.

- **Event Reporting**

  Your agent is responsible for detecting changes in the application's database (or working with the back-end application or its database to be notified when changes occur) and reporting add, change, and delete events to zones via a SIF_Event message. This is done by preparing an `Event` object and passing it to the `Zone.reportEvent` method on the zone to which the data is published.

- **Managing SIF RefIds**

  Your agent is responsible for keeping track of the SIF RefIds that it receives for objects or that it creates for new objects. Although the ADK provides a utility function to generate GUIDs, it does not provide a repository for RefIds storage. This application-specific functionality is the responsibility of your agent.

Java Packages

The ADK is comprised of hundreds of Java classes and interfaces organized into 18 packages. Most core classes are found in the `openadk.library` package, while the remaining packages represent SIF Data Object (SDO) classes organized by SIF Working Group for consistency with the SIF Specification.

Browse the Javadoc to learn more about each package and the classes and interfaces they contain.

Package / Contents	
`openadk.library`	Core classes and interfaces of the OpenADK. Here you'll find essential classes like ADK, ADKException, Agent, AgentProperties, Event, Query, SIFDataObject, SIFMessageInfo, and Zone, as well as message handler interfaces *Subscriber*, *Publisher*, *ReportPublisher* and *QueryResults*,
`openadk.library.assessment`	Classes to model Assessment Working Group Objects. Here you'll find classes like Assessment, AssessmentPackage, and AssessmentForm.
`openadk.library.common`	Classes to model Common Elements. Here you'll find classes like Name, PhoneNumber, and Demographics
`openadk.library.dw`	Classes to model Data Warehouse & Reporting Working Group Objects. Here you'll find classes like AggregateStatisticInfo, AggregateStatisticFact, and AggregateCharacteristicInfo
`openadk.library.etranscript`	Classes to model Student Record Exchange Working Group Objects. Here you'll find classes like StudentRecordExchange, StudentDemographicRecord, and StudentAcademicRecord
`openadk.library.food`	Classes to model Food Services Working Group Objects. Here you'll find classes like StudentMeal, StaffMeal, and FoodserviceItem
`openadk.library.gradebook`	Classes to model Grade Book Working Group Objects. Here you'll find classes like GradingAssignment, MarkInfo, and StudentPeriodAttendance.
`openadk.library.hrfin`	Classes to model Human Resources & Financials Working Group Objects. Here you'll find classes like EmployeePersonal, LocationInfo, and Payment
`openadk.library.infra`	Classes to model SIF Infrastructure Objects (primarily for internal use by the ADK). Some of the ADK's public interfaces pass an instance of the SIF_Error class as a method parameter, so often times you will import this package to access that class.
`openadk.library.instr`	Classes to model Instructional Services Working Group Objects. Here you'll find classes like Activity, Assignment, and Lesson
`openadk.library.library`	Classes to model Library Automation Working Group Objects. Here you'll find the LibraryPatronStatus class.
`openadk.library.log`	Classes and interfaces for the ADK's logging infrastructure. (Note: The classes

Package / Contents
found here are currently for server-side logging only. The open source Apache Log4j framework handles all client logging. In the future, client logging will be abstracted and placed in this package so that developers can use other logging frameworks besides Log4j.)
`openadk.library.profdev` Classes and interfaces to model Professional Development Working Group Objects. Here you'll find the EmployeeCredential and EmployeeCredit classes, among others.
`openadk.library.programs` Classes and interfaces to model Special Programs Working Group Objects. Here you'll find the StudentParticipation and StudentPlacement classes, among others.
`openadk.library.reporting` Classes and interfaces to model Vertical Reporting Task Force Objects. Here you'll find the ReportManifest, StudentLocator, and SIF_ReportObject classes
`openadk.library.student` Classes and interfaces to model Student Information Working Group Objects. Here you'll find classes like StudentPersonal, SchoolInfo, and StudentSchoolEnrollment
`openadk.library.tools` Optional high-level classes for Configuration Files, Mappings, Query Formatters, Load Balancers, and other tools commonly used in the development of agents but outside the scope of the core ADK Class Framework
`openadk.library.tools.cfg` The AgentConfig class implements an XML-based configuration file parser that agents can use as-is or as the basis for custom parsers
`openadk.library.tools.mapping` The Mappings class simplifies the task of mapping field values between the local application's data structures and SIF Data Objects
`openadk.library.tools.metadata` The ADKMetadata class allows custom SIF objects to be defined for use with the ADK.
`openadk.library.tools.queries` The QueryFormatter class simplifies the task of converting SIF_Query elements to other forms such as SQL statements
`openadk.library.tools.xpath` The SIFXPathContext class allows XPath queries to be run against ADK SIF Data Objects
`openadk.library.trans` Classes to model Transportation & Geographic Information Working Group Objects. Here you'll find classes like BusInfo, BusRouteDetail, and BusRouteInfo
`openadk.util` String, GUID, and XML utility routines

Core Classes and Interfaces

The following classes and interfaces are used most frequently in ADK programming. Each is described in greater detail throughout this Guide.

ADK

The `openadk.library.ADK` class represents the ADK library as a global resource of your agent. It is used to initialize the class framework and to define settings that affect the ADK's behavior at runtime. You cannot create an instance of this class directly; rather, a global singleton is established when the `ADK.initialize` method is called. The initialize method is important because it informs the class framework of the version of SIF you will be using and the SIF Data Object – or SDO – libraries to load into memory. Call this method early in your agent startup code.

The ADK class is covered in Chapter 4

Agent

The `openadk.library.Agent` class represents your SIF Agent. It is the central "application" class of the ADK and generally houses the main program entry point. Agents derive a class from this one and call its methods to handle startup and shutdown tasks and to set global properties that are inherited by each zone the agent connects to.

The Agent class is covered in Chapter 5

Zones

The concept of a SIF Zone is central to the architecture of the Schools Interoperability Framework. A zone is a logical partition in which application integration takes place between two or more SIF Agents. Each zone is managed by a Zone Integration Server. The ADK has the ability to connect to more than one zone at the same time.

Common zone topologies include School Zones, which publish data from a single school; District Zones, which publish data from a single school district; and Aggregate Zones, which publish data from a combination of sources. The administrator of a zone integration server will want the flexibility to create as many zones as needed to model data in a school, district, or agency; the more flexible your agent is in terms of zone topologies the better chance it has of working in any SIF environment. If you restrict your agent to publishing data only to a single "district zone", for example, it may not work well with other vendors' agents that are restricted to consuming data from a single "school zone". Data Solutions strongly encourages developers to support the three types of zones mentioned above for maximum flexibility and scalability.

Zones are represented in the ADK by the `openadk.library.Zone` interface. During agent initialization, agents obtain one or more *Zone* instances for each SIF Zone the agent will connect to by calling the methods of the `ZoneFactory` class. The "zone factory" is responsible for creating new *Zone* instances and keeping track of the zones the agent communicates with during a session. You can obtain a reference to the global `ZoneFactory` by calling the `Agent.getZoneFactory` method.

Since *Zone* is an interface implemented privately by the class framework, it cannot be sub-classed. With version 1.1 and later of the ADK, you can associate your own "zone class" with a *Zone* instance by calling the `Zone.setUserData` method, and then later retrieve that class with the `getUserData` method and casting the return value to your class type. This is a common way of organizing your code by zone.

Topics

Depending on the requirements of your application, it may make sense to structure the design of the agent code such that SIF messages are processed in a *data-centric* versus *zone-centric* fashion. For example, you may wish to process messages related to StudentPersonal objects in the same way regardless of the zone from which those messages originate.

The ADK introduces a concept familiar to publish/subscribe frameworks but missing from the Schools Interoperability Framework infrastructure: the *topic*. Data-centric agents may perform all publish, subscribe, and query activity via topics, which serve to aggregate messaging activity across multiple zones. *Topic* instances are obtained from the agent's `TopicFactory`. Each *Topic* represents a single type of SIF Data Object, such as StudentPersonal, LibraryPatronStatus, or BusInfo. By "joining" topics with one or more zones, SIF_Event and SIF_Request messages received from those zones are dispatched to the topic for processing. In this way, your agent's message handling logic can be centralized and organized in a data-centric rather than a zone-centric way.

Zones and Topics are covered in Chapter 6

Agent and Transport Properties

The ADK defines a rich set of operational properties that can be configured programmatically or at runtime with the `/D` Java command-line option. Some properties are used to determine basic agent functionality—such as whether Push or Pull mode is used to communicate with the Zone Integration Server—while others let you fine-tune how the class framework implements the SIF standard. Properties may be set on a zone-by-zone basis; global properties inherited by all zones are defined by the agent class.

Agent and Transport Properties are covered in Chapter 6

SIF Data Objects (SDO) Libraries

The ADK models all SIF Data Objects as object-oriented Java classes. These classes, collectively referred to as the SIF Data Objects libraries, or SDO, exist for top-level object types such as StudentPersonal, LibraryPatronStatus, and BusInfo; for common elements like Name, Address, and PhoneNumber; and for enumerated types, dates, and times. All approved objects from each version of SIF are represented, and in many cases unapproved draft objects are also implemented for early-adopter testing.

A major benefit to the SDO library is its support for parsing and rendering data objects using multiple versions of SIF. For example, an ADK agent can automatically parse and render a SIF 1.0r1 LibraryPatronStatus or SIF 1.1 LibraryPatronStatus object —which differs between the two versions of the specification—without any intervention on your part.

Publishers

A *Publisher* is a message handler class supplied by your agent to process `SIF_Request` messages. *Publisher* is a Java interface that can be implemented by any class of your choosing—typically from your application's data layer.

By registering one or more *Publishers* with your agent's zone and topic classes, you direct the ADK to dispatch incoming `SIF_Request` messages to your application code for processing. Agents typically respond to a `SIF_Request` by querying records in a local database, then converting those records to one or more SIF Data Objects. You can work with SIF Data Objects programmatically by constructing instances of `SIFDataObject` classes supplied for each object in SIF, or you can take a more data-driven approach by using static XPath-style mappings read from a configuration file and managed by the `Mappings` utility class. When responding to `SIF_Requests`, these objects are sent to an output stream supplied by the class framework, which then packages them into individual `SIF_Response` messages that are persisted to disk and ultimately delivered to the zone integration server.

ReportPublisher is a special form of the *Publisher* interface for working with SIF Vertical Reporting objects. Vertical Reporting is available in SIF 1.5 and later.

Publishing with the ADK is covered in Chapter 7

Subscribers

A *Subscriber* is a message handler class supplied by your agent to process `SIF_Event` messages. *Subscriber* is a Java interface that can be implemented by any class of your choosing—typically from your application's data layer.

By registering one or more *Subscribers* with your agent's zone and topic classes, you direct the ADK to dispatch incoming `SIF_Event` messages to your application code for processing. Agents typically respond to `SIF_Event` messages by updating corresponding records in the local application database using the element and attribute values in the SIF Data Object contained in the event payload. You can work with SIF Data Objects programmatically using the `SIFDataObject` classes supplied for each object in SIF, or you can take a more data-driven approach by using static XPath-style mappings read from a configuration file and managed by the `Mappings` utility class.

Subscribing with the ADK is covered in Chapter 8

Queries and QueryResults

Zone and topic classes provide methods to send `SIF_Request` queries to a zone or a specific agent in a zone to request SIF Data Objects. The results of a query are received asynchronously as `SIF_Response` messages. The class framework dispatches these messages to a *QueryResults* message handler as they are received from the zone integration server.

Agents typically process `SIF_Response` messages by converting the SIF Data Objects in the response to database records and importing those records into a local application database. The class framework supplies your message handler a stream from which you can read an arbitrarily large number of `SIFDataObject` instances from the query response. You can work with SIF Data Objects programmatically using the `SIFDataObject` classes supplied for each object in SIF, or you can take a more data-driven approach by using static XPath-style mappings read from a configuration file and managed by the `Mappings` utility class.

Querying with the ADK is covered in Chapter 10

SIF Data Objects Library

The SIF Data Objects (SDO) library is a set of object-oriented Java classes that model all data objects, common elements, and enumerated types defined by the SIF spec. Data objects like StudentPersonal and LibraryPatronStatus are encapsulated by classes of the same name; common elements such as Name, Address, and Demographics are also represented by their own classes. Enumerated type classes are provided wherever a SIF code or type attribute is required. The SDO classes handle proper message rendering and parsing according to the rules of each version of SIF.

The SDO classes are not strictly required for ADK programming. You could work with SIF data in pure XML form using strings or standard interfaces like DOM. There are significant benefits to using the SDO classes, however; they include:

- Automatic message parsing
- Automatic message rendering
- Automatic handling of differences in each version of SIF
- Support for concurrent versions of SIF
- Automatic trimming of `SIF_Response` payloads
- Automatic packetizing of `SIF_Response` messages
- `SIF_Response` messages are returned in the version of SIF requested

Organization of SIF Data Object Classes

The SIF Data Object classes are arranged into Java packages along the same organizational boundaries as SIF Working Groups. Agents can select which packages to include at deployment time, or can choose to load all packages into memory indiscriminately.

For example, if your agent uses objects from Student Information Systems and Food Services but not from other working groups like Transportation or Library Automation, you can choose to distribute only the `sdostudent.jar` and `sdofood.jar` libraries. When initializing the class framework with the static `ADK.initialize` function, instruct the ADK to use only these two modules by passing the `SIFDTD.SDO_STUDENT` and `SIFDTD.SDO_FOOD` flags to that function (logically OR the two flags together).

```
 // Initialize the ADK to use the Student Information Systems and
 // Food Services SIF Data Object modules. You can include sdoall.jar
 // on the classpath, or only the sdostudent.jar and sdofood.jar files
 // to keep the size of your agent to a minimum.

ADK.initialize( SIFVersion.LATEST, SIFDTD.SDO_STUDENT | SIFDTD.SDO_FOOD );
```

Similarly, to include all SIF Data Object modules with your agent, redistribute the `sdoall.jar` file and pass the `SIF.ALL_SDO_LIBRARIES` flag to the `ADK.initialize` function. NOTE: calling the `ADK.initialize()` overload that takes no parameters is equivalent to using the `SIFVersion.LATEST` and `SIFDTD.SDO_ALL` parameters.

```
// Initialize the ADK to use all SIF Data Object modules. Include
// sdoall.jar on the classpath.

ADK.initialize( SIFVersion.LATEST, SIFDTD.SDO_ALL );
```

Java Packages

The table below summarizes the various SIF Data Object packages. Package names are relative to `openadk.library`:

SIF Working Group	openadk.library.* Package	Library File
Common Elements	`common`	*Built-In*
Assessment	`assessment`	`sdoassessment.jar`
Data Warehousing	`dw`	`sdodw.jar`
Student Record Exchange	`etranscripts`	`sdoetranscripts.jar`
Food Services	`food`	`sdofood.jar`
Grade Book	`gradebook`	`sdogradebook.jar`
HR & Financials	`hr`	`sdohrfin.jar`
SIF Infrastructure	`infra`	*Built-In*
Instructional Services	`instr`	`sdoinstr.jar`
Library Automation	`library`	`sdolibrary.jar`
Professional Development	`profdev`	`sdoprofdev.jar`
Special Programs	`programs`	`sdoprograms.jar`
Vertical Reporting & Student Locator	`reporting`	`sdoreporting.jar`
Student Information	`student`	`sdostudent.jar`
Transportation	`trans`	`sdotrans.jar`

Note the SIF Infrastructure data objects (e.g. `SIF_ZoneStatus`) and Common Element data object (e.g. `Name`, `Address`, `Demographics`) are part of the core ADK class framework and thus are included with the `adk-library-<locale>.jar` library file. They're always available regardless of which SDO modules you choose to load when calling the `ADK.initialize` method.

Working with SIF Data Objects

SIFElement & SIFDataObject Classes

All SIF elements modeled by the ADK are encapsulated by classes derived from a common abstract base class: `openadk.library.SIFElement`. These include top-level data objects like StudentPersonal as well as complex child elements like Name, Demographics, Transaction, and so on. (A *complex element* is any element that has attributes or children. In contrast, a simple *field element* is any element that has only a text value. The ADK does not model field elements as Java classes, but it does supply classes to encapsulate each and every complex element found in the SIF data model.)

Classes that model top-level SIF Data Objects like StudentPersonal are derived from the `SIFDataObject` base class, which descends from `SIFElement`. There are over 115 such data objects in the SIF 2.0r1 Specification.

The full class hierarchy is as follows:

```
openadk.library.Element
   openadk.library.SIFElement
      Complex Element Classes
   openadk.library.SIFDataObject
      Data Object Classes
```

Class Constructors

Each `SIFElement` class offers two constructors: a default constructor that accepts no parameters, and another that accepts all required or mandatory elements defined by the SIF Specification. The following code illustrates how to use both forms of constructor. In the first example, a StudentPersonal object is created using that class's default constructor. None of the object's required attributes will have values (in this case the RefId attribute), so it is the responsibility of the agent to ensure they are assigned values before published to SIF.

In the second example, an Address instance is created using the constructor that accepts a value for each required attribute or element (in this case, AddressType, Street/Line1, City, StateProvince, Country, and PostalCode are all denoted as Mandatory in the SIF Specification.) Using this constructor is a shortcut to explicitly assigning values to required attributes and elements, and makes it easy to create a valid Transaction instance in a single line of code:

```
// Create a StudentPersonal object
StudentPersonal sp = new StudentPersonal();
sp.setRefId( ADK.makeGUID() );

// Create an Address
Address addr = new Address(
  AddressType.MAILING, new Street( "321 Baker Dr" ),
  "Metropolis", StatePrCode.IL, CountryCode.US, "98855" );
```

Dynamic SIFDataObject Construction

`SIFDataObject` instances may also be created dynamically by passing an `ElementDef` constant to the `SIFDTD.createSIFDataObject` method, identifying the kind of object to create. (`SIFDTD` and `ElementDef`—two classes central to the SIF Data Objects libraries—are discussed in subsequent sections.)

```
// Kinds of objects to create
ElementDef[] kinds = new ElementDef[] {
    StudentDTD.STUDENTPERSONAL,
    StudentDTD.STAFFPERSONAL,
    LibraryDTD.LIBRARYPATRONSTATUS };

// Create a bunch of SIFDataObject instances...
SIFDTD metadata = ADK.DTD();
SIFDataObject[] objects = new SIFDataObject[ kinds.length ];
for( int i = 0; i < kinds.length; i++ )
  objects[i] = metadata.createSIFDataObject( kinds[i] );
```

Creating & Manipulating SIFDataObjects

The ADK supplies two ways to create and manipulate `SIFDataObject` instances. The first is to programmatically construct objects and call the methods of the SDO classes to get and set element and attribute values. For example,

```
//Constructing a StudentPersonal object
StudentPersonal sp = new StudentPersonal();
sp.setRefId( ADK.makeGUID() );
sp.setName( NameType.LEGAL, "Johnson", "Clifford" );

// Examining a StudentPersonal object
Name n = sp.getName();
System.out.println( "Name: " + n.getLastName() + ", " + n.getFirstName() );
OtherIdList ids = sp.getOtherIdList();
System.out.println( "This student has " + ids.size() + " IDs" );
```

You can also use the setElementOrAttribute method:

```
// Dynamically constructing a StudentPersonal object
SIFDataObject sp = ADK.DTD().createSIFDataObject( StudentDTD.STUDENTPERSONAL );
sp.setElementOrAttribute( "@RefId", ADK.makeGUID() );
sp.setElementOrAttribute( "Name[@Type='01']/LastName", "Johnson" );
sp.setElementOrAttribute( "Name[@Type='01']/FirstName", "Clifford" );

// Dynamically examining a StudentPersonal object
Element refId = sp.getElementOrAttribute( "@RefId" );
Element studentId = sp.getElementOrAttribute( "OtherId[@Type='06'" );
```

An alternative and more data-driven approach to working with `SIFDataObjects` is to use the `Mappings` class from the `openadk.library.tools.Mappings` package to convert `SIFDataObject` instances to and from a data source such as a `Map` by applying a set of XPath-like mapping rules. These rules are usually stored in an external configuration file where they can be modified by system integrators in the field. For example, the following rules for StudentPersonal objects define how to translate the elements and attributes of that object to a flat list of field values:

```
<object object=”StudentPersonal”>
       <field name=”STUDENT_ID”>OtherId[@Type=’01’]</field>
       <field name=”LAST_NAME”>Name[@Type=’01’]/LastName</field>
       <field name=”FIRST_NAME”>Name[@Type=’01’]/FirstName</field>
</object>
```

You can then call upon the `Mappings` class to convert a StudentPersonal instance into a `Map` of field/value pairs, or to convert a `Map` into a StudentPersonal object.

In practice, most agents use a combination of these two approaches when working with SIF Data Objects. Refer to the chapter on SIF Data Objects for more information.

The SIFDTD Class

In order to support all versions of the SIF specification, the ADK has a built-in metadata dictionary. This metadata encompasses information about each element and attribute from all versions of SIF 1.0r1 and later. The class framework relies on this information to determine the versions of SIF each element or attribute supports, its tag name, its sequence number, and various flags such as whether an element is repeatable or not.

The metadata dictionary is comprised of the static constants defined by a DTD class located within each package in the ADK that contains objects representing SIF elements. Each DTD class defines static `ElementDef` constants representing each element and attribute of SIF. `ElementDef` constants identify elements and attributes in a *version-independent* way so that the class framework knows what your agent is referring to regardless of whether the task at hand involves SIF 1.1, SIF 1.5r1, SIF 2.0r1 or some future version of specification.

Whenever a method requires an `ElementDef` parameter, you must pass a constant from one of the package-specific DTD classes:

```
// Create a Topic instance for "BusInfo" and "StudentPersonal" objects
Topic businfo = getTopicFactory().getInstance( TransDTD.BUSINFO );
Topic students = getTopicFactory().getInstance( StudentDTD.STUDENTPERSONAL );
```

SIFDTD Constants for Elements & Attributes

For elements that are part of the *SIF Common Objects* group or that otherwise are used across more than one SIF Data Object, each DTD class uses a naming convention to distinguish between the objects a common element appears in.

For example, the common `<Name>` element is used in StudentPersonal, StaffPersonal, and StudentContact. However, the characteristics of `<Name>`— such as its sequential order relative to its parent—are different when used as a child of `<StudentPersonal>` than when used as a child of `<StaffPersonal>` and `<StudentContact>`. Thus, the `StudentDTD` class defines multiple `ElementDef` constants for the `<Name>` element:

```
StudentDTD.STUDENTPERSONAL_NAME
StudentDTD.STAFFPERSONAL_NAME
StudentDTD.STUDENTCONTACT_NAME
```

The naming convention employed is the name of the parent SIF Data Object in upper-case, followed by an underscore ("_") and the name of the element or attribute, also in upper-case:

```
[DTDClass].PARENT_CHILD
```

For example:

```
// Query for all BusInfo objects
Query q = new Query( TransDTD.BUSINFO );

// Include only the RefId, VehicleNumber, and Contractor in the response
q.setFieldRestrictions(
  new ElementDef[] {
    TransDTD.BUSINFO_REFID,
    TransDTD.BUSINFO_VEHICLENUMBER,
    TransDTD.BUSINFO_CONTRACTOR
  }
);
```

Inspecting the ElementDef of a SIFDataObject

You can obtain the `ElementDef` associated with any `SIFDataObject` instance by calling its `getElementDef` method. This is often used in Boolean comparisons to determine if an object is of a certain type:

```
SIFDataObject someObject = ...

if( someObject.getElementDef() == StudentDTD.STUDENTPERSONAL )
{
  // This is a StudentPersonal object
  StudentPersonal sp = (StudentPersonal)someObject;
  System.out.println("StudentPersonal with RefId " + sp.getRefId() );
}
else
if( someObject.getElementDef() == SIFDTD.STUDENTCONTACT )
{
  etc.
```

Getting an Element's Tag Name

The tag name of an element or attribute is specific to each version of SIF. With the ADK, each `ElementDef` is associated with two names: a version-independent name, which is used to identify the element or attribute regardless of the version of SIF you're working with; and a version-dependent tag name, which is used when rendering and parsing the element. Often times it is necessary to use the element tag name in your code; for example, to log a debug message or to lookup an entry in a table that is keyed by element name.

If you want to obtain the version-*independent* name of an element or attribute, call the `ElementDef.name` method. This method returns the string used to identify the element or attribute in the ADK's metadata dictionary. By convention it is usually equivalent to the name of the element as it first appears in the SIF Specification as of SIF 2.0 or later. .

For example, calling the `name` method on the `LibraryDtd.TRANSACTION` constant returns the string "Transaction":

```
  // Create a Transaction element and show its version-independent name
  Transaction trans = new Transaction();

  System.out.println( trans.getElementDef().name() );
```

If you want to obtain the version-*dependent* name of an element or attribute—that is, the string used when parsing and rendering the element—call the `ElementDef.tag` method. This method requires that a `SIFVersion` instance be passed to it. Unlike the `name` method, it returns the tag name specific to the version of SIF. For example, calling `tag` on the `LibraryDTD.TRANSACTION` constant yields different results for SIF 1.1 than for SIF 2.0. In SIF 1.1, the element is known as "CircTx" and in SIF 2.0 it is known as "Transaction".

```
// Create a Transaction element and show its version-dependent tag name
Transaction trans = new Transaction();

// Show the tag name for SIF 1.1: "CircTX"
System.out.println( trans.getElementDef().tag( SIFVersion.SIF11 ) );

// Show the tag name for SIF 2.0: "Transaction"
System.out.println( trans.getElementDef().tag( SIFVersion.SIF20 ) );
```

The above code uses `SIFVersion` constants but in practice you obtain the `SIFVersion` associated with a SIF Data Object by calling the `SIFDataObject.getSIFVersion` method:

```
  public void onEvent( Event event, Zone zone, MessageInfo info )
    throws ADKException
  {
    SIFMessageInfo inf = (SIFMessageInfo)info;

    // Print the version of the SIF_Message envelope
    System.out.println("Received a SIF_Event from a SIF " +
      inf.getSIFVersion() + " agent" );

    // Print the version-dependent tag name of the data contained in the event
    SIFDataObject data = event.getData().readDataObject();
    System.out.println( "A " + data.getElementDef().tag( data.getSIFVersion() )
+
      " object is contained in the event" );
    System.out.println( "The " + data.getElementDef().name() + " object is a
SIF " +
      data.getSIFVersion() + " object" );
```

The above example might print the following to the Java console:

```
Received a SIF_Event from a SIF 1.1 agent
A StudentPersonal object is contained in the event
The StudentPersonal object is a SIF 1.1 object
```

Building SIF Data Objects Dynamically

Most agents retrieve their data from a local database or other data store with its own unique schema and programming interfaces. A common approach to interacting with SIF is to dynamically *map* elements and attributes from a SIF Data Object to fields in the local application's database. The ADK provides a powerful facility for mapping: the `Mappings` class found in the `openadk.library.tools.mapping` package. You could also implement your own mapping routines by using functions such as `SIF-DataObject.setElementOrAttribute`.

The following code creates a StudentPersonal object by enumerating the fields in a mapping table. This code looks much different from most of the code shown throughout the Developer's Guide and ADK Example agents because it does not directly use the SIF Data Object classes like `Name`, `PhoneNumber`, and `StudentAddress` to build the StudentPersonal object. Instead, it uses XPath-like query strings and the `setElement-`

`OrAttribute` method of the `SIFDataObject` class. With the ADK, you have the flexibility to work with SIF Data Objects however you like.

Refer to the `SIFDataObject` class in the Javadoc for more information on the `setElementOrAttribute` method.

```
// Build a table that maps fields in the local application to SIF
// elements & attributes. The key of each entry is a name of your
// choosing that identifies a field in the local database; the value
// of each entry is an XPath-link query string that identifies
// the corresponding SIF element or attribute of the StudentPersonal
// object

HashMap m = new HashMap();
m.put( “ForeignId”,    “@RefId” );
m.put( “ID”,           “OtherId[@Type=’06’]” );
m.put( “L_Name”,       “Name[@Type=’01’]/LastName” );
m.put( “F_Name”,       “Name[@Type=’01’]/FirstName” );
m.put( “Addr_Line1”,   “StudentAddress/Address[@Type=’M’]/Street/Line1” );
m.put( “Addr_Line2”,   “StudentAddress/Address[@Type=’M’]/Street/Line2” );
m.put( “Addr_City”,    “StudentAddress/Address[@Type=’M’]/City” );
m.put( “Addr_State”,   “StudentAddress/Address[@Type=’M’]/StatePr/@Code” );
m.put( “Addr_Zip”,     “StudentAddress/Address[@Type=’M’]/PostalCode” );
m.put( “Pri_Email”,    “Email[@Type=’Primary’]” );
m.put( “Sec_Email”,    “Email[@Type=’Alternate1’]” );

// Now create a StudentPersonal from the mapping table. For each entry
// in the table, lookup the associated value in the local application
// database, then call the StudentPersonal.setElementOrAttribute method
// to assign that value to the corresponding element or attribute in the
// SIF Data Object

StudentPersonal student = new StudentPersonal();
for( Iterator it = m.keySet().iterator(); it.hasNext(); )
{
  String localField = (String)it.next();
  String sifField = (ElementDef)m.get( field );

  // Lookup the value of this field from the local application’s database
  String value = myDatabaseView.getFieldValue( localField );

  // Assign the field to the StudentPersonal object
  if( value != null )
    student.setElementOrAttribute( sifField, value );
}
```

Enumerated Types

The ADK defines enumerated type classes whenever a set of pre-defined values is expected for an object field. These classes are named the same as the attribute or element they’re associated with. For example, the `EducationLevel/Code` element of the `StudentContact` data type is represented by the `EducationLevelCode` class.

As you might expect, the `StudentContact.setEducationLevel` method accepts a single parameter: an `EducationLevelCode` object. Valid codes for this object are defined as static constants of the enumerated type class: `EducationLevelCode.EMC`, `EducationLevelCode.E4`, etc.

Enumerated type classes are provided as a convenience to programmers, to enforce type safety, and to help ensure that valid values are used in the construction of SIF messages. There are many cases when referencing one of the pre-defined constants of an enumerated type class is not appropriate: for example, when reading values from a database result set you will want to assign them directly to an element or attribute. For this reason, all enumerated type classes offer a `wrap` method that you can use to pass your own string in place of a pre-defined constant. The following code demonstrates:

```
// Construct a StudentContact object
StudentContact sc = new StudentContact();

// Use the enum class to set EducationLevel to “E4”
sc.setEducationLevel( EducationLevelCode.E4 );

// Set the code directly
sc.setEducationLevel( EducationLevelCode.wrap( “E4” ) );

// Can also set the code to an invalid value
sc.setEducationLevel( EducationLevelCode.wrap( “Zebra” ) );
```

Dates and Times

SIF 2.0 uses XSD data types to represent Dates, DateTimes, and Times. The ADK represents each of these fields using the Java Calendar object.

Creating RefIds for SIF Data Objects

When an agent publishes a data object to a zone for the first time, it must assign the object a globally-unique identifier—or *RefId*—that will forever identify that object in the zone. Both the `ADK.makeGUID` static function and `Agent.makeGUID` convenience method create Globally Unique Identifiers (GUIDs).

```
// Create a new StudentPersonal object with a new GUID generated
// by this agent
StudentPersonal sp = new StudentPersonal();
sp.setRefId( ADK.makeGUID() );
...
```

Managing RefIds

Because it is specific to your agent’s transaction layer and involves interaction with a database or other persistent data store, the ADK does not provide any classes to manage RefIds. You will need to create a mechanism to store RefIds created by your agent when publishing SIF Data Objects for the first time, or imported from other agents when receiving objects in SIF_Response and SIF_Event messages. Although there are a number of ways to go about this, one simple solution is to use a database table to record the RefIds known to your agent. The following schema will usually suffice:

Column	Data Type	Definition
RefId	Char[32]	The GUID
PrimaryKey	String or Numeric	The primary key of the record in your application's database that is associated with this RefId.
SecondaryKey	String or Numeric	The optional secondary key of the record in your application's database that is associated with this RefId.
ObjectType	Numeric	A constant that identifies the type of data associated with this RefId. Consider defining numeric values for each object type; for example, 1=Students or StudentPersonal; 2=Teachers or StaffPersonal; etc.

When publishing an object to a zone for the first time, assign a RefId to it and record the association between that RefId and the local application database record. You must use this RefId to subsequently identify the object whenever it is included in a SIF message: for example, when reporting a SIF_Event to a zone.

When receiving an object in a SIF_Event or SIF_Response message, the agent should first lookup its RefId in the above table. Most SIF Data Objects have an attribute named "RefId" that you can obtain via the `SIFDataObject.getRefId` method. If the RefId exists in your table, take the appropriate action on the corresponding database record. For example, if you receive a SIF_Event with a Change action that identifies a student you have on file, update the student record appropriately. However, if you receive a Change event for a student that does not exist in your RefId repository, you cannot take action on the event because the agent doesn't know which student record to update.

How does the RefId repository get created initially? Data Solutions recommends that all agents incorporate some form of "synchronization procedure" that is run the first time the agent is installed and again at the beginning of each school year. This process retrieves all SIF Data Objects from the zone or zones the agent is connected to, and using an algorithm of your choosing, matches them up with corresponding records in the application database. The goal of synchronization is to establish an initial set of RefIds such that your agent knows which SIF Data Objects equate to the record that already exist in the application's database. Once this process is complete, you can begin responding to SIF Events.

There are many ways to implement a synchronization procedure, some more involved than others. For example, you might have a user interface for the administrator to control the synchronization process, to choose the kinds of SIF Data Objects to query from the zone, and to provide a way for the administrator to visually match up application records with their corresponding SIF Data Objects. Or, your might opt for a more automated process where the agent requests objects from the zone and tries to match them with application records using a common field such as Student ID. Regardless of the approach you take, be sure to include some form of synchronization procedure in your agent's design.

Parsing & Rendering SIF XML

The ADK is designed to automatically parse and render SIF XML for you at the appropriate times. For example, when an incoming SIF_Event message is received, the data contained within is parsed into a `SIFDataObject` instance and passed to the *Subscriber* message handler registered with the Zone or Topic the message was received on.

The `SIFParser` and `SIFWriter` classes are used internally to accomplish all parsing and rendering.

The SIFWriter Class

You can use the `SIFWriter` class to programmatically render SIF XML for your own needs. For example, suppose you have an instance of a `openadk.library.student.StudentPersonal` object in memory and want to render it to Java's `System.out` stream or to a text file. Use the `SIFWriter` class to wrap an output stream and write any `SIFDataObject` instance to that stream in SIF XML format:

```
// Construct a StudentPersonal object
StudentPersonal myStudent = new StudentPersonal();
myStudent.setRefId( ADK.makeGUID() );
myStudent.setName( new Name( “Smith”, “Joe”) );

// Render it to System.out
SIFWriter writer = new SIFWriter( System.out, true /* auto-flush */ );
writer.write( myStudent );

// Also render it to a file named MyStudent.xml
FileOutputStream fos = new FileOutputStream( “MyStudent.xml” );
writer = new SIFWriter( fos, true );
writer.write( myStudent );
fos.close();
```

The above code writes a `<StudentPersonal>` element to the Java console as well as to a file named MyStudent.xml.

The SIFParser Class

Use the `SIFParser` class to wrap an input stream and parse any SIF XML into a `SIFElement` instance (of which `SIFDataObject` is a sub-class). Note `SIFParser` is intended to parse *data objects* only and not full `SIF_Message` content.

The following code streams in the previous section's MyStudent.xml file from disk and parses it back into a `StudentPersonal` object:

```
// Construct a SIFParser instance that can be used for parsing
SIFParser p = SIFParser.newInstance();

// Open a FileReader to read the text file
FileReader in = new FileReader( "MyStudent.xml" );

// Read the file into a StringBuffer in 4K chunks
StringBuffer xml = new StringBuffer();
int bufSize = 4096;
char[] buf = new char[ bufsize ];
while( in.ready() ) {
  bufSize = in.read( buf, 0, buf.length );
  xml.append( buf, 0, bufsize );
}

// Parse it and cast the result to a StudentPersonal object
StudentPersonal student = p.parse(xml.toString(), null /* zone */ );
System.out.println( "Read student from disk; RefId is "+student.getRefId() );
```

Encoding XML Character Entities

In XML, certain characters like the less-than and greater-than symbols, apostrophes, and the ampersand must be encoded (or "escaped") with special character entities. For example, the string "Red & Blue" must be encoded as "Red &amp; Blue"; the name "Kim D'Angelo" must be encoded as "Kim D&apos;Angelo", and so on. XML parsers will automatically unencode these strings so they're presented to receiving agents as "Red & Blue" and "Kim D'Angelo", respectively.

The `SIFWriter` class in ADK 1.5.1 and later **automatically** encodes all XML content according to the table below; there is nothing special you need to do for this to happen.

Invalid Character	Equivalent XML Entity
`<`	`&lt;`
`>`	`&gt;`
`&`	`&amp;`
`'`	`&apos;`

IMPORTANT: Since the ADK handles encoding all XML content for you, its important that agents do not also perform this task. Otherwise "double-escaping" can occur. For example, consider the string "Kim D'Angelo". The apostrophe is invalid and must be encoded as "&apos;". Suppose the agent were to perform this on its own and assigned a resulting value of "Kim D&apos;Angelo" to an element. When rendering the element, the ADK's SIFWriter class would also attempt to encode it. Since the ampersand character is invalid, the string would be incorrectly rendered as "Kim D&amp;apos;Angelo" instead of "Kim D&apos;Angelo". The result? Receiving agents would parse the string and receive a student named "Kim D&aposAngelo" instead of "Kim D'Angelo", which is obviously not the desired outcome.

Disabling XML Encoding

There are some cases when it is desirable to turn off escaping for specific elements. For example, suppose you intend to write in-line XML content to a `SIF_ExtendedElement` element. Automatic encoding in this case is not desirable because all of the < and > characters in your XML content would be escaped. In these rare cases you can use the `setDoNotEncode` method to turn off encoding on an element-by-element basis:

```
StudentPersonal sp = new StudentPersonal();

// Get the SIF_ExtendedElements container
SIF_ExtendedElements container = sp.getSIF_ExtendedElementsContainer();

// Construct a SIF_ExtendedElement with in-line XML content, but turn off
// encoding by calling the setDoNotEncode method
SIF_ExtendedElement inlineXml = new SIF_ExtendedElement( “teaminfo”,
  “<![CDATA[<team>Blue &amp; Red</team>]]” );
inlineXml.setDoNotEncode( true );

// Add the element to the container
container.addChild( inlineXML );
```

SIF Infrastructure Messages

One aspect of ADK programming you might find surprising is that the class framework handles sending SIF Infrastructure messages for you at the appropriate times, and generally does not allow you to send these messages yourself. A good example is the sending of `SIF_Ack` messages. The ADK acknowledges messages behind the scenes after it has dispatched a message to a message handler, but does not offer you a way to directly send a `SIF_Ack` message to a zone.

There are ways you can influence how and when infrastructure messages are sent—and there is even a limited API for sending them yourself—but in general this task should be left up to the ADK. The following table summarizes the methods that directly or indirectly cause SIF Infrastructure messages to be sent to the Zone Integration Server.

Message	When the message is sent
`SIF_Ack`	Sent by the ADK upon receiving a message from the Zone Integration Server. `SIF_Ack` is sent after calling your agent’s message handler. If an exception is thrown by the message handler, a `SIF_Ack` containing an error is returned; otherwise a success `SIF_Ack` is returned. When using the `TrackQueryResults` class to issue a SIF_Request query while processing a `SIF_Event`, the ADK may invoke Selective Message Blocking (SMB) on the event by sending a `SIF_Ack` with the appropriate status code. A second `SIF_Ack` message is sent to clear the SMB state as soon as your message handler returns control to the ADK.
`SIF_Event`	Sent by the ADK when the `reportEvent` method is called on a Zone.
`SIF_Provide`	Sent by the ADK when the `Zone.connect` method is called and you have registered a *Publisher* with one or more Zones or Topics. The `ADKFlags.PROV_PROVIDE` flag must be passed to the `Zone.setPublisher` or `Topic.setPublisher` method for this message to be sent. If you agent wishes to respond to `SIF_Request` messages but does not want to register as a Provider with the zone, specify the `ADKFlags.PROV_NONE` instead.
`SIF_Register`	Sent by the ADK when the `Zone.connect` method is called and you have specified the `ADKFlags.PROV_REGISTER` flag.

Message	When the message is sent
	It is acceptable to send a `SIF_Register` message each time your agent starts up because the Zone Integration Server will return a successful status code.
`SIF_Request`	Sent by the ADK when the `Zone.query` or `Topic.query` method is called, or when the `TrackQueryResults` class is used to issue a synchronous query.
`SIF_Response`	Sent by the ADK when data is returned by your `Publisher-.onRequest` message handler. The ADK handles dividing response data into packets, so more than one `SIF_Response` message may be sent if the amount of data to return is larger than the maximum buffer size of the requesting agent.
`SIF_Subscribe`	Sent by the ADK when the `Zone.connect` method is called and you have registered a *Subscriber* with one or more Topics or Zones. The `ADKFlags.PROV_SUBSCRIBE` flag mus be passed to the Zone.`setSubscriber` or `Topic.setSubscriber` method for this message to be sent.
`SIF_SystemControl/ SIF_Sleep`	Sent by the ADK when the `Zone.sleep` or `Agent.sleep` method is called. Also sent when the `Agent.shutdown` method is called.
`SIF_SystemControl/ SIF_Wakeup`	Sent by the ADK when the `Zone.wakeup` or `Agent.wakeup` method is called. (Note that the sending of a `SIF_Register` message on agent startup will automatically wake up a sleeping agent per SIF Specifications.)
`SIF_SystemControl/ SIF_Ping`	Sent by the ADK when the `Zone.connect` or `Zone.sifPing` method is called. The ADK always sends a `SIF_Ping` message when connecting to a zone in order to determine if the agent is sleeping or not.
`SIF_SystemControl/ SIF_GetMessage`	Sent periodically by the ADK when running in Pull mode to retrieve the next group of messages waiting in the agent's queue on the Zone Integration Server. The polling frequency defaults to 30 seconds but may be changed by setting the `adk.messaging.pullFrequency` agent or zone property.
`SIF_Unprovide`	This message is not sent automatically by the ADK. To manually send it, call the `Zone.sifUnprovide` method.
`SIF_Unsubscribe`	This message is not sent automatically by the ADK. To manually send it, call the `Zone.sifUnsubscribe` method.
`SIF_Unregister`	Sent when the `Agent.shutdown` or `Zone.disconnect` method is called and the `ADKFlags.PROV_UNREGISTER` flag is passed to that method.

Working with SIF Messages

Message Handlers

A message handler is a callback class that implements one of the ADK's four message handler interfaces:

Interface	Handles
`openadk.library.Publisher`	SIF_Request messages
`openadk.library.ReportPublisher`	SIF_Request messages for Ver-

	tical Reporting objects
`openadk.library.Subscriber`	SIF_Event messages
`openadk.library.QueryResults`	SIF_Response messages

Message handlers form the cornerstone of asynchronous message processing in the ADK. Because they're Java interfaces, they can be implemented on any class of your choosing and are easily combined into existing code. This interface-based approach results in clean, organized, understandable code that works the same regardless of whether you're connected to one zone or a hundred.

Classes that implement the message handler interfaces are responsible for processing `SIF_Request`, `SIF_Event`, and `SIF_Response` messages received by the agent. Message processing is always performed asynchronously and may begin as early as the `Zone.connect` method is called on a given zone, so be sure to implement your handlers with thread synchronization in mind.

You can register a message handler with a `Topic` object, a `Zone` object, or the `Agent` object. Agents that use topic classes typically register handlers with *Topic* instances only. If you aren't using topics, you register handlers with each `Zone` instance. These three classes—Topic, Zone, and Agent—each provide the following methods for registering message handlers:

- `setPublisher( Publisher handler, ElementDef objType, PublishingOptions flags )`
- `setReportPublisher( ReportPublisher handler, ReportPublishingOptions flags )`
- `setSubscriber( Subscriber handler, ElementDef objType, SubscriptionOptions flags )`
- `setQueryResults( QueryResults handler, ElementDef objType, QueryResultsOptions flags )`

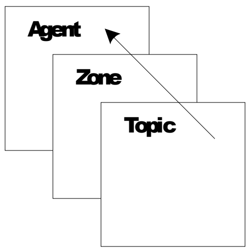
The *objType* parameter is a constant from the `SIFDTD` class that identifies the type of SIF Data Object the message handler will service. You can call the above methods repeatedly for different kinds of SIF Data Objects, passing the same message handler instance to each call. The following code demonstrates how to register the same *Subscriber* class to handle `SIF_Events` received from a zone for StudentPersonal, SchoolInfo, and StudentSchoolEnrollment objects:

```
// Get a zone instance from the Agent’s ZoneFactory
Zone myZone = getZoneFactory().getZone( “Zone1”,
   “http://localhost:7080/Zone1” );

SubscriptionOptions subOpts = new SubscriptionOptions();

// Register SIF_Event message handlers
myZone.setSubscriber( this, StudentDTD.STUDENTPERSONAL,
    subOpts );
myZone.setSubscriber( this, StudentDTD.SCHOOLINFO,
    subOpts );
myZone.setSubscriber( this, StudentDTD.STUDENTSCHOOLENROLLMENT,
    subOpts );
```

Message Dispatching

Message Handlers work like traditional callback functions: the class framework uses them to call into your code when a SIF message is received and needs to be processed. It calls the appropriate message handler asynchronously whenever a `SIF_Event`, `SIF_Request`, or `SIF_Response` message is received from a zone. When there is more than one message handler for a zone, the type of SIF Data Object contained in the message payload is used to determine which one to call. This is why you must specify an `ElementDef` constant to the `setPublisher` and `setSubscriber` methods of the Zone and Topic interfaces.

Recall that you can set message handlers on the Agent, Zone, and Topic classes. These three classes form a “message dispatching chain”:



When an incoming `SIF_Message` is received, the ADK first attempts to dispatch it to the *Topic* instance corresponding to the type of object contained in the message. For example, if a `SIF_Event` is received with StudentPersonal objects, the class framework first tries to dispatch the message to the *Subscriber* message handler registered with the `SIFDTD.STUDENTPERSONAL` topic. (This of course assumes your agent is using topic classes, which are entirely optional.)

If no *Topic* is found to handle the message, the ADK next consults the *Zone* object from which the message originated. If a message handler hasn’t been registered with the zone object, the ADK makes one final attempt at dispatching the message directly to the Agent class. These three entities—topic, zone, and agent—form the message dispatching chain by which all incoming messages are sent for processing.

When to Implement Message Handler Interfaces

An agent may implement one or more message handlers on as many classes as is appropriate. Trivial agents, such as the ADK Examples, might choose to implement all interfaces directly on the `Agent` class:

```
public class MyAgent extends Agent implements Publisher, Subscriber
```

More complex agents typically implement message handlers on multiple classes. For example, consider an agent with a database layer comprised of classes to encapsulate each table in the database. Such an agent might implement the *Publisher* and *Subscriber* message handlers on each of these database classes, thereby placing the knowledge of publishing and subscribing near the database layer:

```
public class MyAgent extends Agent;

public class StudentDatabase implements Publisher, Subscriber;

public class EnrollmentDatabase implements Publisher, Subscriber;

public class FoodServiceDatabase implements Publisher, Subscriber;

public class DataImporter implements QueryResults;

etc.
```

Where in your code you choose to implement message handlers all depends on your preference and overall agent design. At Data Solutions, we commonly create our own "SIFZone" class that implements all of the message handler interfaces in one place. Then we attach this class to the ADK *Zone* instance by calling the `Zone.setUserData` method (see also Attaching User Data to a Zone on page 64):

```
public class SIFZone implements Publisher, Subscriber, QueryResults
{
  // The ADK Zone instance wrapped by this custom SIFZone class
  protected Zone fZone;

  /** Constructor */
  public SIFZone( Zone adkZone ) {
    fZone = zone;
    fZone.setUserData( this );
  }

  /** Set up the zone and its message handlers */
  public void setup() {
    fZone.setSubscriber( this, StudentDTD.STUDENTPERSONAL,
      new SubscriptionOptions() );
    ...
    fZone.setPublisher( this, LibraryDTD.LIBRARYPATRONSTATUS,
      new SubscriptionOptions() );
    ...
    fZone.setQueryResults( this );
  }

  /** Connect to the zone */
  public void connect() {
    setup();
    fZone.connect( ADKFlags.PROV_REGISTER );
    ...
  }

  etc.
```

Registering Message Handlers with Zones

Call the `setPublisher`, `setSubscriber`, and `setQueryResults` methods of the *Zone* interface to register a *Publisher*, *Subscriber*, or *QueryResults* implementation with a zone. Any messages received from the zone will then be dispatched to your message handler according to the object type contained in the payload:

```
// Create a Zone instance
Zone myZone = getZoneFactory().getZone( “Zone1”,
   “http://localhost:7080/Zone1” );

// Register message handlers for this zone
myZone.setSubscriber( database, StudentDTD.STUDENTPERSONAL,
      new SubscriptionOptions() );
myZone.setSubscriber( database, StudentDTD.STUDENTSCHOOLENROLLMENT,
      new SubscriptionOptions() );

myZone.setQueryResults( database, StudentDTD.STUDENTPERSONAL );
myZone.setQueryResults( database, StudentDTD.STUDENTSCHOOLENROLLMENT );

myZone.setPublisher( database, StudentDTD.LIBRARYPATRONSTATUS,
    new PublishingOptions );
```

In this example, *Subscriber* and *QueryResults* message handlers are registered with the zone to handle incoming `SIF_Event` and `SIF_Response` messages that contain StudentPersonal and StudentSchoolEnrollment payloads. The `PROV_SUBSCRIBE` flag causes the ADK to send a `SIF_Subscribe` message to the zone when it connects. A *Publisher* message handler is also registered for LibraryPatronStatus objects; the `PROV_PROVIDE` flag instructs the ADK to send a `SIF_Provide` message for this object type when connecting to the zone.

Registering Message Handlers with Topics

When a message handler is registered with a *Topic*, it is called by the class framework for messages associated with the topic’s object type regardless of the zone from which the message originated. Thus, you can write code that is data-oriented instead of zone-oriented.

```
Topic students = getTopicFactory().getTopic(
    SIFDTD.STUDENTPERSONAL );
Topic enrollments = getTopicFactory().getTopic(
    SIFDTD.STUDENTSCHOOLENROLLMENT );
Topic library = getTopicFactory().getTopic(
    SIFDTD.LIBRARYPATRONSTATUS );

// Register message handlers for these topics
students.setSubscriber( database, ADKFlags.PROV_SUBSCRIBE );
enrollments.setSubscriber( database, ADKFlags.PROV_SUBSCRIBE );
students.setQueryResults( database );
enrollments.setQueryResults( database );
library.setPublisher( database, ADKFlags.PROV_PROVIDE );


// Join zones to the topic and connect to each zone...
```

The Subscriber Interface

Classes that implement the *Subscriber* interface handle `SIF_Event` messages received by the agent. This message handler is covered in greater detail by the Subscribing chapter beginning on page 75.

The *Subscriber* interface defines the following method:

```
public void onEvent( Event event, Zone zone, MessageInfo info )
  throws ADKException;
```

The Publisher Interface

Classes that implement the *Publisher* interface handle `SIF_Request` messages received by the agent. This message handler is covered in greater detail by the Publishing chapter beginning on page 65.

The *Publisher* interface defines the following method:

```
public void onRequest(
  DataObjectOutputStream out,
  Query query,
  Zone zone,
  MessageInfo inf )
    throws ADKException;
```

The ReportPublisher Interface

Classes that implement the *ReportPublisher* interface handle `SIF_Request` messages where the object type specified in the request is a SIF_ReportObject. Only agents that implement SIF 1.5 Vertical Reporting need to implement this interface. It must be used in place of the standard *Publisher* interface if you plan to respond to requests for SIF_ReportObject.

The *ReportPublisher* interface defines the following method:

```
public void onReportRequest(
  String reportObjectRefId,
  ReportObjectOutputStream out,
  Query query,
  Zone zone,
  MessageInfo inf )
    throws ADKException;
```

The QueryResults Interface

Classes that implement the *QueryResults* interface handle `SIF_Response` messages received by the agent. This message handler is covered in greater detail by the Querying chapter beginning on page 81.

The *QueryResults* interface defines the following two methods:

```
public void onQueryPending( MessageInfo info, Zone zone )
  throws ADKException;
public void onQueryResults(
  DataObjectInputStream data,
  SIF_Error error,
  Zone zone,
  MessageInfo info )
    throws ADKException;
```

Throwing Exceptions from Message Handlers

All message handlers may throw an exception of type `com.edustructure.sif-works.ADKException` if an error occurs while processing a message. When you throw an exception from a message handler, the ADK returns an error acknowledgement to the ZIS in the form of a `SIF_Ack` message. The error category, error code, description, and extended description fields of the `SIF_Ack/SIF_Error` element depend on the type of exception thrown. When a message handler returns successfully, the ADK acknowledges the message with a success `SIF_Ack`.

Exceptions are handled slightly differently for `SIF_Request` messages because, according to SIF Specifications, errors that occur during request & response handling should be returned to the requestor as a `SIF_Error` element in the payload of a `SIF_Response` packet. When the ADK catches an exception thrown from the `Publisher.onQuery` method, it returns a *success* `SIF_Ack` for the original `SIF_Request` message. A `SIF_Error` element is then embedded in the last `SIF_Response` packet delivered to the server.

Provisioning SIF Zones

The Schools Interoperability Framework requires that an agent be registered with a SIF Zone in order to send messages. It also requires that an agent declare its intent to subscribe to SIF Data Objects or to serve as the zone's authoritative provider of an object type. We refer to these tasks collectively as "provisioning". Provisioning involves the sending of `SIF_Register`, `SIF_Unregister`, `SIF_Subscribe`, `SIF_Unsubscribe`, `SIF_Provide`, and `SIF_Unprovide` messages.

ⓘ Note: The SIF Specification refers to "provisioning" as the act of sending SIF_Provide and SIF_Unprovide messages. We use the term to encompass all SIF registration messages sent by an agent, including SIF_Register, SIF_Provide, SIF_Subscribe, SIF_Unregister, SIF_Unprovide, and SIF_Unsubscribe.

SIF_Register & SIF_Unregister Messages

A `SIF_Register` message is automatically sent to the Zone Integration Server when the `Zone.connect` method is called and the `ADKFlags.PROV_REGISTER` flag is passed to that method. Agents therefore register each time they start up (assuming your code is written to connect to zones at startup time), which ensures the registration data on the server is always up-to-date. You can manually send a `SIF_Register` message by calling the `Zone.sifRegister` method.

A `SIF_Unregister` is sent when the `Zone.disconnect` method is called and the `ADKFlags.PROV_UNREGISTER` flag is passed to that method. Using this flag is not generally recommended because un-registering an agent clears its queue and all state information from the Zone Integration Server. You can manually send a `SIF_Unregister` message by calling the `Zone.sifUnregister` method.

SIF_Subscribe & SIF_Unsubscribe Messages

Agents that subscribe to `SIF_Event` messages must send a `SIF_Subscribe` message to the zone integration server for each object type they wish to subscribe to. The server will then distribute events to the agent as they're received from other agents in the zone.

The ADK handles subscription for you when the `Zone.connect` method is called. It determines the list of objects to subscribe to by examining the *Subscriber* message handlers that have been registered with the zone. If the `ADKFlags.PROV_SUBSCRIBE` flag was passed to the `setSubscriber` method, a `SIF_Subscribe` message is sent for that object. If this flag is not used, the ADK does not automatically send a `SIF_Subscribe` message.

It is important that agents register message handlers with Zones and Topics *before* calling `Zone.connect` to ensure the appropriate provisioning messages are sent.

The ADK does not automatically send `SIF_Unsubscribe` messages to zones. You can do this manually by calling the `Zone.sifUnsubscribe` method.

SIF_Provide & SIF_Unprovide Messages

Agents that wish to be the authoritative or default provider of an object type in a zone must send a `SIF_Provide` message for each object type. This instructs the zone integration server to route undirected `SIF_Requests` to the agent. Note only one agent in a zone can be designated the provider of an object type. This does not preclude other agents from responding to `SIF_Request` messages explicitly directed at them, so you can still write *Publisher* message handlers for objects even if your agent is not assuming the role of provider.

The ADK handles sending SIF_Provide messages for you when the `Zone.connect` method is called. It determines the list of objects by examining the *Publisher* message handlers that have been registered with the zone. If the `ADKFlags.PROV_PROVIDE` flag was passed to the `setPublisher` method, a `SIF_Provide` message is sent for that object. If this flag is not used, the ADK does not automatically send a `SIF_Provide` message.

It is important that agents register message handlers with Zones and Topics *before* calling `Zone.connect` to ensure the appropriate provisioning messages are sent.

The ADK does not automatically send `SIF_Unprovide` messages to zones. You can do this manually by calling the `Zone.sifUnprovide` method.

Working with Multiple Versions of SIF

The ADK supports all versions of SIF 1.0r1 and later. Agents developed with the ADK can operate in mixed environments where not all agents connected to a zone use the same version of SIF. In most cases, you do not need to worry about SIF versioning because the ADK handles the details for you. For example, when a `SIF_Request` is received the ADK formats your responses to conform to the version of SIF specified by the requestor.

Default Version

Although the ADK supports multiple versions of SIF simultaneously, you need to declare the “default version” your agent will use. This is done at agent startup when initializing the class framework. Unless you specify otherwise, the “default version” is the latest version of the SIF Specification supported by the ADK. This is a global setting that applies to all zones to which the agent is connected for the current session.

All outbound messages generated by your agent will be rendered according to the default version. For inbound messages, the ADK inspects the *xmlns* and *Version* attributes of the `SIF_Message` envelope to decide which version to use when parsing the message. If the version of a message is supported by the ADK, the message is parsed accordingly regardless of the default version you declared when you initialized the ADK. Thus, agents can operate in mixed environments because they are able to receive and process messages from any sender.

If the ADK receives a message for a version of SIF it does not support, an error `SIF_Ack` is returned to the sender:

```
<SIF_Error>
  <SIF_Category>12</SIF_Category>
  <SIF_Code>2</SIF_Code>
  <SIF_Desc>SIF 2.0r6 is not supported</SIF_Desc>
  <SIF_ExtendedDesc>http://www.sifinfo.org/v2.0r6/messages</SIF_ExtendedDesc>
</SIF_Error>
```

How the ADK Registers Multiple Versions with a Zone

When the ADK sends a `SIF_Register` message to a zone, it declares the versions of SIF the agent will support by including one or more `SIF_Version` elements in the registration message. These elements let the Zone Integration Server know which versions of the specification your agent is capable of supporting, as well as the version it considers to be its default. The server can then avoid sending messages to the agent if it does not support those messages.

The ADK uses several factors in determining how to prepare the `SIF_Version` elements of the `SIF_Register` message:

- The default version of SIF used by the agent
- The versions of SIF supported by the ADK
- The latest version of SIF supported by the Zone Integration Server

Note that with the SIF 1.1 specification released in early 2003, agents may declare support for more than one version of the standard. However, earlier versions of SIF restricted the agent to declaring only a single version number. To avoid incompatibilities with earlier Zone Integration Servers that do not yet support SIF 1.1, the ADK takes the following approach to registration:

- It assumes the Zone Integration Server supports SIF 1.1 or later and therefore will accept registration messages that use SIF 1.1 conventions. If this is not the case, call the `AgentProperties.setZisVersion` method to inform the ADK of the latest version supported by the Zone Integration Server. To ensure the server accepts its registration messages, the ADK will prepare the `SIF_Register` message to conform to this version of SIF.

- The ADK declares each version of SIF it supports in the `SIF_Register` message
- The ADK declares the agent's default version of SIF to be the version passed to the `ADK.initialize` method at agent startup. By default, this value is the latest version supported by the ADK

To illustrate, consider an agent that has initialized the ADK to use SIF 1.0r1 as the agent's default version, but is connecting to a Zone Integration Server that supports SIF 1.1. The class framework will send the following `SIF_Register` message to the server using a SIF 1.1 message envelope, declaring all versions of the specification supported by the ADK, and declaring the default version to be SIF 1.0r1:

```
<SIF_Message xmlns="http://www.sifinfo.org/infrastructure/1.x" Version="1.1">
  <SIF_Register>
    ...
    <SIF_Version>1.0r1</SIF_Version>
    <SIF_Version>1.0r2</SIF_Version>
    <SIF_Version>1.1</SIF_Version>
    ...
```

If this agent were to call `AgentProperties.setZisVersion("1.0r1")` to inform the ADK that the latest version supported by the Zone Integration Server is 1.0r1, the `SIF_Register` message would be sent in a SIF 1.0r1 message envelope and would only include a single `SIF_Version` element per SIF 1.0r1 specifications:

```
<SIF_Message xmlns="http://www.sifinfo.org/v1.0r1/messages">
  <SIF_Register>
    ...
    <SIF_Version>1.0r1</SIF_Version>
```

Agent Logging

Server Logging

With SIF 1.5 and later, an agent can contribute to the zone integration server's logs by reporting a `SIF_LogEntry` object to a zone via an Add event. `SIF_LogEntry` can be used to log simple error messages, or it can be used to refer to a `SIF_Message` and a set of data objects previously received by the agent. It is designed to be used just like any other object in the Schools Interoperability Framework. `SIF_LogEntry` participates in both the request and response and subscription models of SIF and may have a designated provider agent in each zone, a role that's usually assumed by the zone integration server.

One notable difference between `SIF_LogEntry` and other objects in SIF is that it only supports Add events; you cannot Change or Delete log entries once they've been published to a zone. Another difference is in the way `SIF_LogEntry` events are reported by the ADK. Rather than construct an instance of `SIF_LogEntry` and report an event with the normal `Zone.reportEvent` method, agents call the methods of the `ServerLog` class, available from each zone by calling the `Zone.getServerLog` method.

The ServerLog Class

The `ServerLog` class represents a logical log container on the server, and can be extended and customized to support logging mechanisms other than `SIF_LogEntry`. In addition, it handles some of the more complicated underlying details of constructing `SIF_LogEntry` objects, such as replicating the event header in the object's `SIF_LogEntryHeader` element.

Follow these steps to report a `SIF_LogEntry` to a zone:

1. Call the `Zone.getServerLog` method to obtain the zone's `ServerLog` instance. This class is defined in the `openadk.library.log` package.

2. Call one of the forms of the `ServerLog.log` method to report a log entry to the zone integration server.

The following example reports a simple text message to the server:

```
myZone.getServerLog().log( "Agent started synchronization" );
```

Other forms of the `log` method exist that let you pass more information, such as extended message text and an error category and code:

```
MyZone().getServerLog().log(
    LogLevel.ERROR,
    "Agent has run out of memory and will shut down",
    outOfMemoryException.getMessage(),
    "SYS-100", /* Application-defined error code */
    LogEntryCodes.CATEGORY_ERROR,
    LogEntryCodes.CODE_AGENT_FAILURE );
```

Logging References to SIF Messages & SIF Data Objects

The code examples above show how to write simple messages to the server log. One of the major benefits of the `SIF_LogEntry` object is that it can optionally refer to a previously received `SIF_Message` and set of data objects. This information can be used by consumers of `SIF_LogEntry` to associate log entries with messages and data for message tracking and analysis purposes.

To include a reference to a previous `SIF_Message`, call the form of `ServerLog.log` that accepts a `SIFMessageInfo` parameter. `SIFMessageInfo` is passed to all ADK message handler functions and contains header information that the ADK will use to build the `SIF_LogEntry/SIF_OriginalHeader` element.

The following example shows how to report a `SIF_LogEntry` object that indicates an agent was unable to process a `SIF_Event` because of a failure in the application's business rules. The `SIFMessageInfo` instance passed to the `Subscriber.onEvent` method, which contains header information describing the `SIF_Event` message, is passed to the `log` method. In addition, the subject of the failure, a `StudentPersonal` object, is packaged into an array and passed as the final parameter to that method.

```
// Subscriber message handler
public void onEvent( Event ev, Zone zone, MessageInfo inf )
    throws ADKException
{
    ...
    if( failedObject != null )
    {
        MyZone().getServerLog().log(
            LogLevel.ERROR,
            "Could not delete student John Smith due to a business rule",
            null, /* no extended description */
            null, /* no application-defined error code */
            LogEntryCodes.CATEGORY_DATA_ISSUES_WITH_FAILURE,
            LogEntryCodes.CODE_BUSINESS_RULE_FAILURE,
            (SIFMessageInfo)inf,
            new SIFDataObject[] { failedObject } );
    }
```

Using ServerLog with Versions of SIF Prior to 1.5

The `SIF_LogEntry` object was not introduced into the Schools Interoperability Framework specification until 1.5. When an agent is running in an earlier version of the specification, the `ServerLog` class does not send `SIF_LogEntry` objects to the server. Instead, it writes the error message and code information passed to the `log` method to the local agent log for the zone.

Echoing SIF_LogEntry to the Agent Log

By default, the ADK automatically writes a summary line to the local agent log for the zone whenever the `ServerLog.log` method is called. The summary line includes the following elements from the `SIF_LogEntry` object:

- `SIF_Desc`
- `SIF_ExtendedDesc`
- `SIF_Category`
- `SIF_Code`
- `SIF_ApplicationCode`

The `SIF_OriginalHeader` and `SIF_LogObject` elements, if present, are not included in the summary line.

The echoing of `SIF_LogEntry` messages is generally recommended because it allows administrators to view important log information from either the agent or server logs. Further, there is no guarantee that a zone will consume `SIF_LogEntry` events; it may in fact ignore them if the ZIS does not have internal support for this object type and there are no subscribers in the zone. Nonetheless, you can disable the echo feature by adjusting the settings of the class framework's default *ServerLogModule* implementation module:

1. Obtain the `ServerLog` instance at the top of the class framework's logging chain by calling `ADK.getServerLog`

2. Call the `ServerLog.getLoggers` method to obtain an array of *ServerLogModule* implementations.

3. Enumerate the array. For each instance, call the `getID` method. If the string returned equals "DefaultServerLogModule", proceed to step 4.

4. Cast the *ServerLogModule* instance to `openadk.library.log.DefaultServerLogModule` and call its `setEcho` method with a value of false

The following code illustrates:

```
// Disable the echoing of ServerLog entries to the local agent log
ServerLog root = ADK.getServerLog();
ServerLogModule[] loggers = root.getLoggers();
for( int i = 0; i < loggers.length; i++ ) {
    if(( loggers[i].getID().equals("DefaultServerLogModule") ) {
        ((DefaultServerLogModule)loggers[i]).setEcho( false );
    }
 }
```

The Agent's Work Directory

When the `Agent.initialize` method is called, the ADK creates a *work directory* on the local file system. This directory is used to store `SIF_Response` packets pending delivery to the ZIS as well as messages that are too large to fit in memory. It is also used by the Agent Local Queue when writing messages to the local file system.

The location of the work directory is relative to the agent's home directory, obtained by calling the `Agent.getHomeDir` method. The default implementation of this method chooses a home directory as follows:

- If the `adk.home` System property is set, it is used as the home directory

- If the `adk.home` System property is not set, the `user.dir` System property is used instead. On most Java platforms this property is set to the directory from which the Java program was started.

You may override the `Agent.getHomeDir` method to return an alternate home directory if your agent has unique requirements. If the `Agent.initialize` method cannot verify the work directory exists, or cannot create it, an exception is thrown.

Part III

CORE CLASSES

4. The ADK Class

The `openadk.library.ADK` class represents the ADK library as a global resource of your agent. It is used to initialize the class framework and to set properties that affect the ADK's behavior at runtime. You cannot create an instance of this class directly; instead, it is established automatically when the `ADK.initialize` method is called.

Initializing the Class Framework

Prior to calling any ADK methods or instantiating `SIFDataObject` classes, you must first initialize the class framework. The static `ADK.initialize` method accepts two parameters:

```
public static void initialize(
     SIFVersion defaultVersion,
     int sdoLibraries )
          throws ADKException;
```

Parameter	Description
`defaultVersion`	Identifies the default version of SIF your agent will use. Because it supports multiple versions of SIF concurrently, the ADK is capable of processing incoming messages regardless of version number. However, it requires that you declare a default version to be used for outbound messages originating from the agent. Most of the time you will want to use the latest version supported by the ADK, but on occasion it may be necessary to run your agent as if it was written for an earlier version of the SIF Specification. The default version is a global setting. Refer to Working with Multiple Versions of SIF on page 37 for more information.
`sdoLibraries`	Identifies the SIF Data Object modules to load into memory. Constants are defined by the `openadk.library.SDO` class. To load multiple modules into memory, logically OR the constants; for instance, `(SDO.STUDENT|SDO.LIBRARY)` loads SIF Data Objects

	from both the Student Information Systems working group and the Library Automation working group. If an SDO module is not loaded, any constants from the SIFDTD class defined by the module are null and cannot be used at runtime. You will receive IllegalStateExceptions when trying to reference objects from an SDO module that is not loaded.

For example,

```
// Initialize the ADK to use SIF 1.1 and to load the SIS and Food
// Services SDO modules
ADK.initialize( SIFVersion.SIF_11, SDO.STUDENT | SDO.FOOD );
```

Calling the `ADK.initialize` method with no parameters uses the latest version of SIF supported by the class framework and loads all SDO modules loaded into memory:

```
public static void initialize()
  throws ADKException;
```

When to Initialize

Initialization is typically done early on in your agent's startup routine—in the program's `main` function or in `Agent.initialize`. If you perform ADK initialization from your `Agent.initialze` method, be sure to do so *before* calling the superclass implementation.

For example,

```
public class MyAgent extends Agent
{
  public void initialize()
  {
    // Initialize the ADK
    ADK.initialize();

    // Now call the superclass
    super.initialize();
```

Debug Output

The ADK features rich debugging support via the Apache Log4J library. By default, minimal debugging is enabled when the ADK is initialized. You may adjust this setting at agent startup depending on the configuration file or command-line options of your agent by modifying the `ADK.debug` value directly:

```
// Turn off debugging
ADK.debug = ADK.DBG_NONE;

// Turn on minimal debugging
ADK.debug = ADK.DBG_MINIMAL;

// Turn on only logging of Exceptions and detailed messaging traces
// with content
ADK.debug =  ADK.DBG_EXCEPTIONS |
             ADK.DBG_MESSAGING_DETAILED |
             ADK.DBG_MESSAGING_CONTENT;
```

We suggest experimenting with the ADK's various debug flags so you can see what kinds of messages are written to the log when each is enabled. It's also a great way to learn what the ADK is doing under the hood as it sends and receives SIF messages. Each of the example agents look for a "`/debug n`" option on the command-line, where *n* is a debugging level in the range 0 through 5. Try running the example agents with different debug levels turned on.

Redirecting Debug Output to a Log File

The default output for debug messages is the Java console. To redirect all log output to a log file, call the static `ADK.setLogFile` method or set the `adk.log.file` System property prior to ADK initialization. This can be done programmatically as shown below, or by specifying the `/D` option on the Java command-line.

```
// Redirect log output to "agent.log"
System.setProperty( "adk.log.file", "agent.log" );

// Initialize the ADK
ADK.initialize();
```

From the Java command-line:

```
java /Dadk.log.file=c:/temp/myagent.log ...
```

Other Ways to Redirect Debug Output

Because the ADK uses Apache's Log4J library as its underlying log library, you can customize that system to redirect output to the Windows NT Event Viewer, to a relational database, or to any other "appender" class for the Log4J environment. The ADK class framework defines several static `Category` objects that you can access and manipulate with the Log4J API:

- To obtain a reference to the ADK's `Category` instance, reference the static `ADK.log` variable directly; the class framework writes debug messages to this Category prior to agent initialization

- To obtain a reference to the Agent's `Category` instance, call the `Agent.getLog` method; the class framework writes global, zone-independent debug messages to this Category

- To obtain a reference to a Zone's `Category` instance, call the version of the `Agent.getLog` method that accepts a *Zone* parameter; the class framework writes zone-specific debug messages to these Categories

Other ADK Static Methods

Most of the other public methods of the ADK class are used internally by the class framework. Refer to the Javadoc for a complete description of each.

Method	Description
`getADKVersion`	Gets the version of the ADK library
`getSIFVersion`	Gets the default version of SIF in effect for the agent. This is the `SIFVersion` argument passed to the `initialize` method, or if none was provided, the latest version of SIF supported by the class framework.
`getSupportedSIFVersion`	Enumerates all SIF versions supported by the class framework
`getTransportProtocols`	Enumerates all transport protocols supported by the class framework
`initialize`	Initializes the ADK
`isInitialized`	Determines if the ADK has been initialized
`isSIFVersionSupported`	Determines if a SIF version is supported by the class framework
`makeGUID`	Creates a Globally Unique Identifier
`setLogFile`	Redirects log output to a file

Summary

- The ADK class represents the class framework as a resource of your application
- You cannot create instances of the ADK class; all of its methods are static
- Call the `ADK.initialize` method prior to using other classes in the framework
- Set the debugging flag and optionally redirect debugging to a log file prior to initializing the class framework

5. The Agent Class

The `openadk.library.Agent` class represents your SIF Agent to the framework and typically houses the main program entry point. Derive a class from `Agent` and override the `initialize` method to perform all initialization and startup tasks, such as setting global agent properties, optionally establishing *Topics*, and connecting to zones.

To use the `Agent` class,

- Derive a class from `openadk.library.Agent`
- Create an instance of your `Agent` class
- Call the `Agent.initialize` method

Constructor

The `Agent` constructor requires that you pass the *SourceId* by which your agent will be identified in all messages it generates. The SIF Specification requires that each agent in a zone have a unique *SourceId* value.

```
public class MyAgent extends Agent
{
  public MyAgent()
  {
    this( "MyCompanyAgent" );
  }

  public MyAgent( String sourceId )
  {
    super( sourceId );

    setName( "Descriptive Agent Name" );
  }
```

The *SourceId* of your agent should be a configurable value so that administrators can change it if necessary—for instance, in the unlikely event another vendor has selected the same default *SourceId* as you have, or the administrator wants to run multiple copies of the agent. You may call the `Agent.setSourceId` method to change the value after construction, but this should be done *prior* to connecting to any zones. Changing the *SourceId* after your agent has connected to zones will result in errors from the Zone Integration Server.

Agent Initialization

Before using an instance of the `Agent` class, it must be initialized:

```
// Create and initialize the Agent object
MyAgent agent = new MyAgent();
agent.initialize();
```

The initialize method is a great place to centralize your own agent startup tasks. Agents typically override this method to:

- Set default agent properties (e.g. Push or Pull mode)
- Set default transport protocol settings (e.g. the port number for Push mode)
- Optionally establish one or more *Topics*
- Connect to one or more zones

The remainder of this chapter covers each of these tasks in greater detail. You may want to refer to the ADK Example agents for a working example—use these agents as a starting point for creating your own.

When overriding the `initialize` method, make sure to call the superclass:

```
public class MyAgent extends Agent
{
  public void initialize() throws Exception
  {
    super.initialize();

    // Set default properties
    AgentProperties props = getProperties();
    props.setMessagingMode( AgentProperties.PUSH_MODE );

    // Set transport protocol properties
    ...
```

Agent Properties

The ADK defines a fairly large set of operational properties that can be configured programmatically or at runtime with the /D Java command-line option. Some properties are used to control basic SIF functionality—such as whether Push or Pull mode should be used to communicate with the Zone Integration Server, what port to listen on for Push mode, and so on—while others let you fine-tune how the class framework implements the SIF standard.

ADK properties are encapsulated by the `AgentProperties` class. There is one instance of this class for the agent and one instance for each zone. To access the default `AgentProperties` object, call `Agent.getProperties`. Next, call the get and set methods of this object to change the default properties inherited by all zones.

```
// Enable Pull mode with a polling frequency of 1 minute
AgentProperties props = myAgent.getProperties();
props.setMessagingMode( AgentProperties.PULL_MODE );
props.setPullFrequency( 60000 );
```

Most properties can be set on a zone-by-zone basis as well. To access the `AgentProperties` object of a specific zone, call `Zone.getProperties`. The property values set on this object apply only to the zone. For example, to instruct the class framework to use Push mode for one zone in particular,

```
// Create the “district” zone and instruct it to use Push mode
Zone dz = getZoneFactory().getInstance(
   “DistrictZone”, “http://209.11.65.3:7500/DistrictZone” );
AgentProperties zoneProps = dz.getProperties();
zoneProps.setMessagingMode( AgentProperties.PUSH_MODE );

// Also change default security settings for this one zone
zoneProps.setEncryptionLevel( 3 );
zoneProps.setAuthenticationLevel( 3 );
```

Specifying Properties on the Java Command-Line

When the `Agent.initialize` method is called, the ADK scans all Java system properties for entries that begin with the “`adk.`” prefix. If any are found, they’re assigned to the `AgentProperties` object as defaults. By using the /D option on the Java command-line, you can dynamically adjust the ADK’s settings at runtime without making any changes to code or configuration files.

For example, the following command-line changes the messaging mode to Push:

```
java /Dadk.messaging.mode=Push –classpath ...
```

Common Properties

Refer to the Javadoc for a comprehensive list of settings defined by the AgentProperties class. Some of the more commonly-used properties are summarized below. Those marked with ◆ are global and cannot be defined on a zone-by-zone basis.

Property Name / Default Value	Description
◆`adk.defaultTimeout` Default: `30000` ms	The default timeout value used by the ADK when the agent does not provide a specific value
◆`adk.messaging.effectiveBufferSize` Default: `32000` bytes	The internal buffer size used by the class framework; determines when data is off-loaded to disk
`adk.messaging.keepMessageContent` Default: "`false`"	When true, the `SIFMessageInfo.getMessage` method returns the XML content of the message as it was received by the Zone Integration Server. When false, the class framework does not preserve message content and therefore returns a null when this method is called.
`adk.messaging.maxBufferSize` Default: `20000000` bytes	The `SIF_MaxBufferSize` value used by the agent when sending `SIF_Register` and `SIF_Request` messages
`adk.messaging.mode` Default: "`Push`"	The messaging mode used to communicate with the Zone Integration Server; may be set to either "Push" or "Pull"
`adk.messaging.pullFrequency` Default: `30000` ms	Determines how often the ADK polls the Zone Integration Server for messages when running in Pull mode
`adk.messaging.transport` Default: "`http`"	The transport protocol used to communicate with the Zone Integration Server.
`adk.messaging.sleepOnDisconnect` Default: "`true`"	Determines if a `SIF_SystemControl/SIF_Sleep` message is sent to a zone when the `Zone.disconnect` or `Agent.shutdown` method is called
`adk.provisioning.mode` Default: "`adk`"	The Provisioning Mode in effect for the agent. The class framework defines three such modes: ADK-managed, Agent-managed, and ZIS-managed
`adk.security.authenticationLevel` Default: `0`	The minimum authentication level that must be in effect by any channel over which the agent's messages are allowed to travel. The ADK uses this value when constructing SIF message headers (i.e. the `SIF_Header/SIF_Authentication-Level` element)
`adk.security.encryptionLevel`	The minimum encryption level that must

Property Name / Default Value	Description
Default: `0`	be in effect by any channel over which the agent's messages are allowed to travel. The ADK uses this value when constructing SIF message headers (i.e. the `SIF_Header/SIF_EncryptionLevel` element)

Of all properties, three are changed most often:

- `adk.messaging.mode`
- `adk.messaging.pullFrequency`
- `adk.transport.protocol`

Transport Protocols

The ADK supports two transport protocols by default:

- HTTP
- HTTPS

Additional protocols may be offered in the future.

Prior to connecting your agent to any zones, you should set transport properties as needed. Push-mode agents in particular must set the HTTP/HTTPS port on which to listen for messages. The ADK does not assume a default port. (Pull mode agents do not establish a local port for listening so this is not a requirement when running in Pull mode). In addition, you can change the IP address to listen on if more than one network card is installed. By default, the ADK listens on all addresses.

HTTP is the default transport protocol. To change it, call the `AgentProperties.-setTransportProtocol` method and pass the text name of a protocol supported by the ADK (e.g. "http" or "https"). Note you can obtain the correct name from the `TransportProperties` object as shown below.

```
// Set HTTPS properties
HttpsProperties https = myAgent.getDefaultHttpsProperties();
https.setPort( 9443 );
https.setRequireClientAuth( true );

// Change the default protocol to HTTPS
AgentProperties props = myAgent.getProperties();
props.setTransportProtocol( https.getProtocol() );
```

Transport Properties

Each transport protocol supported by the ADK is assigned a "default" `Transport-Properties` object used by all zones. For HTTP and HTTPS, subclasses exist that make it easier to get and set properties specific to those protocols. You can obtain the default properties with the following methods:

```
// Get the default properties for a protocol (e.g. "http" or "https")
TransportProperties getDefaultTransportProperties( String protocol )
// Get the default HTTP properties
HttpProperties getDefaultHttpProperties();

// Get the default HTTPS properties
HttpsProperties getDefaultHttpsProperties();
```

The latter methods are convenience methods. If you call `getDefaultTransportProperties` directly instead of using one of these convenience methods, you should cast the return value to the appropriate subclass. For example,

```
HttpProperties http = (HttpProperties)getDefaultTransportProperties("http");
```

Setting Default HTTP Properties

To change the default HTTP properties for the agent,

- Obtain the `HttpProperties` object from the `Agent` instance:

```
HttpProperties http = myAgent.getDefaultHttpProperties();
```

- Call the `HttpProperties` setter methods to adjust settings:

```
// Change the push port to 12000 and the push address to 210.1.45.15
http.setPort( 12000 );
http.setHost( "210.1.45.15" );
```

Setting Default HTTPS Properties

Refer to the "HTTPS" section in the Deployment chapter for more information regarding HTTPS.

To change the default HTTPS properties for the agent,

- Obtain the `HttpsProperties` object from the `Agent` instance:

```
HttpsProperties https = myAgent.getDefaultHttpsProperties();
```

- Call the `HttpsProperties` setter methods to adjust settings:

```
https.setPort( 12443 );
https.setKeystore( "agent.cer" );
https.setKeystorePassword( "changeit" );
https.setRequireClientAUth( false );
```

Specifying Properties on the Java Command-Line

You can also define Transport Properties on the command-line using the following naming convention: "`adk.transport.`*`protocol`*`.`*`property`*`=value`", where *protocol* is one of the transport protocols supported by the ADK like "http" or "https". For example, to change the default HTTP port number used by Push mode agents,

```
java /Dadk.transport.https.port=7505 /Dadk.transport.https.keystore=ü
zis.cer /Dadk.transport.https.requireClientAuth=true
```

Customizing the Default Transport Properties for a Zone

The ADK uses the default transport properties whenever it needs to establish a listening socket for Push-mode agents. Thus, the defaults can be used for all zones to which your agent connects.

However, if you want to customize a specific zone with its own transport properties—for example, to require Client Authentication on one zone in particular—you can create your own `TransportProperties` instance and assign it to a zone directly. The ADK will then use your custom properties for that zone instead of choosing the defaults.

When creating a new `TransportProperties` object, you must supply the constructor with an object from which default values can be inherited. Simply pass the default transport properties as illustrated below:

```
// Obtain the default properties for HTTPS
HttpsProperties httpsDefs = getDefaultHttpsProperties();
httpDefs.setClientAuth( false );
httpDefs.setPort( 12443 );
// Create a new TransportProperties instance with custom values
TransportProperties myZoneHttps = new TransportProperties( httpsDefs );
myZoneHttps.setPort( 12444 );
myZoneHttps.setHost( "10.0.0.4" );
myZoneHttps.setClientAuth( true );

// Assign custom properties to myZone
Zone myZone = getZoneFactory().getZone( "myzone" );
myZone.setTransportProperties( myZoneHttps );
```

Agent Shutdown

The `Agent.shutdown` method performs important cleanup tasks:

- Closes resources held internally by the ADK
- Calls the `disconnect` method of each *Zone* instance, which by default sends a `SIF_Sleep` message to prevent the Zone Integration Server from pushing messages while the agent is off-line
- If the `ADKFlags.PROV_UNREGISTER` flag is passed to the `shutdown` method, a `SIF_Unregister` message to each zone to unregister the agent and permanently clear the contents of its queue on the server.

`Agent.shutdown` should always be called when the agent is terminated. You can ensure this by using a `try-finally` block in your main function as illustrated by all ADK Example agents.

Lifecycle Considerations

Although we tend to think of SIF Agents as clients of the Zone Integration Server, agents play the role of a server, too. They should run at all times to process messages received by the ZIS on behalf of the local application, even when the ZIS is not available or the local application is not running.

To build a robust SIF Agent, you should give special attention to its lifecycle:

- Implement the agent's main thread so that it runs even when there is no messaging activity; or, if embedded in another Java application, ensure the agent's thread is allowed to run in the background at all times

- Wrap the agent in an NT Service on Microsoft Windows platforms—or as an equivalent service-level process on other platforms—so that it will launch unattended when the computer is started

- Implement a "Connection Watchdog" to retry the connection of zones when the agent starts up it may be started before the Zone Integration Server, particularly if both are running on the same server and it is rebooted.

- When a critical resource is lost—such as a database connection—call the `Zone.sleep` method to temporarily halt messaging between the agent and that zone. When the resource is recovered, call the `Zone.wakeup` method to restore messaging. Placing an agent in sleep mode not only prevents the Zone Integration Server from attempting to push messages to it, but also signals system administrators that the agent is off-line and may be experiencing a problem.

Preventing the Main Thread from Ending

If your agent runs external to your application, you can prevent the main thread from ending as illustrated in the code below. The approach taken here is two-fold: First, a Java "shutdown hook" is installed so that the agent can detect when the virtual machine is being terminated. This is a feature of Java 1.3 and later. Second, a semaphore is created to block the thread until the shutdown hook is called.

When used in combination, this approach effectively prevents your agent's main thread from ending until Ctrl+C is pressed in the Java console, the `System.exit` method is called, or the operating system requests that the process be terminated.

```
static Object sLifecycle = new Object();

public static void main( String[] args )
{
  MyAgent agent = null;

  try
  {
    // Create and initialize the agent
    agent = new MyAgent();
    agent.initialize();

    // Do whatever else is required by your agent at startup time

    // Create a "shutdown hook" to notify the sLifecycle object
    // when Java is exiting
    final Object _lifecycle = sLifecycle;
    Runtime.getRuntime().addShutdownHook(
      new Thread() {
        public void run() {
          synchronized( _lifecycle ) {
            _lifecycle.notifyAll();
          }
        }
      }
    );

    // Wait for the shutdown hook to notify the sLifecycle object
    synchronized( sLifecycle ) {
      sLifecycle.wait();
    }

    // Agent is now ending...

  }
  finally
  {
    // Ensure Agent.shutdown is always called
    if( myAgent != null ) {
      try {
        myAgent.shutdown();
      } catch( Exception e ) {
        System.out.println( e );
      }
    }
  }
```

Notice a `try-finally` block is created to ensure the agent is shutdown properly. Shutting down the agent is necessary so it can free its local resources (e.g. close the local queues, etc.) and disconnect from zones.

Public Interface

Refer to the Javadoc for a complete description of each method.

Method	Description
`getDefaultHttpProperties`	Gets the default HTTP transport properties. This is a convenience method equivalent to calling `getTransportProperties` with a value of "http" and casting the result to an `HttpTransportProperties` object.
`getDefaultHttpsProperties`	Gets the default HTTPS transport properties. This is a convenience method equivalent to calling `getTransportProperties` with a value of "https" and casting the result to an `HttpsTransportProperties` object.
`getTransportProperties`	Gets the default `TransportProperties` object for a transport protocol supported by the ADK
`getHomeDir`	Gets the agent's home directory
`getId`	Gets the *SourceId* value passed to the constructor. The agent is known by this unique name to all zones with which it is registered.
`getLog`	Gets the Log4J "category" for the agent or for a specific zone
`getName`	Gets the agent product name. By default, this method returns the value returned by `getId`. The ADK calls this method to obtain the `UserAgent` header for HTTP messages sent to Zone Integration Servers. Override this method to specify your own product name.
`getProperties`	Gets the `AgentProperties` object. These are global properties inherited by all zones.
`getPublisher`	Gets the global *Publisher* message handler for a specific SIF object type. This message handler will be called when a SIF_Request message is received but a *Publisher* has not been registered with the source zone or topic.
`getQueryResults`	Gets the global *QueryResults* message handler for a specific SIF object type. This message handler will be called when a SIF_Response message is received but a *QueryResults* has not been registered for the source zone or topic.
`getReportPublisher`	Gets the global *ReportPublisher* message handler for Vertical Reporting agents. This message handler will be called when a SIF_Request message for SIF_ReportObject objects is received but a *ReportPublisher* has not been registered for the source zone or topic.
`getSubscriber`	Gets the global *Subscriber* message handler for a specific SIF object type. This message handler will

	be called when a SIF_Event message is received but a *Subscriber* has not been registered for the source zone or topic.
`initialize`	Initialize the agent. This method must be called prior to calling any other function of the `Agent` class.
`IsInitialized`	Determines if the `initialize` method has been called to initialize the agent.
`isShutdown`	Determines if the `shutdown` method has been called to shut down the agent.
`makeGUID`	Creates a Globally Unique Identifier (a SIF RefId)
`setPublisher`	Registers a global *Publisher* message handler with the Agent class.
`setQueryResults`	Registers a global *QueryResults* message handler with the Agent class.
`setReportPublisher`	Registers a global *ReportPublisher* message handler with the Agent class.
`setSubscriber`	Registers a global *Subscriber* message handler with the Agent class.
`shutdown`	Shutdown the Agent by closing local resources and sending a `SIF_Sleep` message to each zone to which the agent is connected. Agents should always call this method prior to terminating.
`sleep`	Sends a `SIF_Sleep` system control message to each zone to which the agent is connected.
`wakeup`	Sends a `SIF_Wakeup` system control message to each zone to which the agent is connected.

Summary

- The Agent class represents your SIF Agent to the class framework
- Derive a class from `openadk.library.Agent`
- Pass the agent's unique *SourceId* to the constructor
- Override the `initialize` method to set default agent and transport properties; don't forget to call the superclass implementation
- The ADK does not assign a default HTTP or HTTPS port to agents, so you must set this transport property during agent initialization (only applicable when the agent is running in Push mode)
- Call the `initialize` method prior to connecting to zones
- Call the `shutdown` method when the agent process is ending

6. Zones & Topics

About Zones

The concept of a “zone” is central to the architecture of the Schools Interoperability Framework. A zone is a logical entity in which application integration takes place among two or more agents. Each zone is managed by a Zone Integration Server.

With the ADK, you can build agents that have the ability to connect to multiple zones for district-wide installations of SIF.

Zone Objects

Zones are represented by the `openadk.library.Zone` interface. During agent startup, you obtain *Zone* instances for each of the zones your agent will connect to by calling the methods of the `ZoneFactory` class. The zone factory is a global resource of the agent that serves to create new *Zone* objects and to keep track of which zones the agent is connected to. Call the `Agent.getZoneFactory` method to obtain a reference to the zone factory:

```
public void initialize() throws Exception
{
  super.initialize();

  // Obtain Zone instances for each zone the agent will connect to
  ZoneFactory factory = getZoneFactory();
  ...
```

Next, call the `getInstance` method to obtain a *Zone* instance. If this is the first time calling this method for a zone, the factory returns a new *Zone* object. Subsequently, the same object will be returned.

```
  // Obtain Zone instances for each zone the agent will connect to
  ZoneFactory factory = getZoneFactory();
  Zone[] zones = new Zone[] {
    factory.getInstance( “MyZone”, “http://localhost:7080/MyZone” ),
    factory.getInstance( “DistrictZone”, “https://zis.org:7443/DistrictZone” ),
  };
```

We recommend reading a list of zones from a configuration file and providing a graphical user interface from which system administrators can manage zones and optionally specify properties for each, such as whether Push or Pull mode is to be used. Most of the ADK Examples allow you to specify zones in a configuration file as well as on the command-line.

Zone IDs and URLs

The first parameter to the `getInstance` method is the Zone ID. The Zone ID is usually a short string such as “DISTRICT_ZONE” or “Ramsey” that is chosen by the server administrator when configuring the ZIS. When connecting to the ZIS, keep in mind that zone identifiers are case-sensitive and cannot include spaces or special characters like the period, colon, and ampersand.

The second parameter to the `getInstance` method is the zone URL. The ADK connects to the zone by sending an HTTP or HTTPS request to this URL, so it must begin with *http:* or *https:* when these two transport protocols are used. The format of the URL is specific to the ZIS product you're using. With the ZIS, the URL must be in the following format, where *zoneID* is case-sensitive and identifies a zone on the server (e.g. "Ramsey"):

```
http://host:port/zoneID
```

Connecting to Zones

Once a *Zone* instance has been obtained from the `ZoneFactory`, you can establish a connection with the Zone Integration Server. Once connected, the ADK may immediately begin to receive messages from the server so it is important a few setup tasks be performed first:

- If Topics are being used as discussed in the next section, join the zone with one or more topics by calling the `Topic.join` method. Any messages received from the zone will be dispatched to the message handler of the appropriate topic.

- If Topics are not being used—or you want to handle message processing specially for a zone—register *Publisher*, *Subscriber*, and *QueryResults* message handlers directly with the zone instance by calling its `setPublisher`, `setSubscriber`, and `setQueryResults` methods. These message handlers are discussed in chapters 7, 8, and 10.

- If a zone is to have custom properties that differ from the agent's defaults, prepare an `AgentProperties` object and set any zone-specific values. This properties object can then be passed to the `zone.connect` method.

Once the above tasks are complete, call the `Zone.connect` method to establish a connection with the zone. Depending on the flags passed to that method and the message handlers registered with the zone and topic classes, the ADK may send a combination of `SIF_Register`, `SIF_Subscribe`, `SIF_Provide`, and `SIF_Ping` messages at this time.

There are two forms of the `Zone.connect` method:

```
public void connect( int adkFlags )
  throws ADKException;

public void connect( int adkFlags, AgentProperties properties )
  throws ADKException;
```

- The **`adkFlags`** parameter should be set to `ADKFlags.PROV_REGISTER` to send a SIF_Register message to the zone. It is recommended that you always send a registration message because it wakes up the agent if previously sleeping and releases all messages blocked with Selective Message Blocking. To connect to a zone without sending a SIF_Register message, specify the `ADKFlags.PROV_NONE` flag or a value of zero.

  No other `ADKFlags` have an effect on the connect method.

- The **`properties`** parameter defines properties specific to this zone only. By default, a zone inherits the agent's default properties. Refer to the *Zone Properties* section below for more information.

Connection Errors

If the Zone Integration Server returns an error acknowledgement for a registration or provisioning message, the `Zone.connect` method will throw a `SIFException` describing the details of the error. If you connect to zones by looping through an array as illustrated below, make sure to use the try-catch block inside the loop so that the agent will attempt to connect to as many zones as possible:

```
// Obtain Zone instances for each zone the agent will connect to
ZoneFactory factory = getZoneFactory();
Zone[] zones = new Zone[] {
  factory.getInstance( "MyZone", "http://localhost:7080/MyZone" ),
  factory.getInstance( "DistrictZone", "https://zis.org:7443/DistrictZone" ),
};

// TODO: Join the zone to one or more Topics

// TODO: Create custom properties for the zone if applicable

// Now connect to each zone...
for( int i = 0; i < zones.length; i++ )
{
  try
  {
    zones[i].connect( ADKFlags.PROV_REGISTER );
  }
  catch( SIFException se )
  {
    System.out.println( "Could not connect to zone " + zones[i] );
    System.out.println( "The server reported an error: " + se.toString() );
  }
}
```

As with all `SIFExceptions`, you can inspect the error category, code, description, and extended description by calling the `getErrorCategory`, `getErrorCode`, `getErrorDesc`, and `getErrorExtDesc` methods. This is useful if you want to react to a connection error based on the error category and code. To simply print an exception to the console, call the `toString` method (used above) to obtain a string in this style:

```
"[Category=1, Code=1]: Description. Extended Description."
```

Zone Properties

By default, each *Zone* object inherits the default global options of the agent. Recall that the `AgentProperties` object stores default operational settings such as the messaging mode to use to connect to the Zone Integration Server. The properties of the agent, which serve as defaults for all zones to which the agent connects, are obtained by calling the `Agent.getProperties` method.

You can also set properties on a zone-by-zone basis. There are two ways to accomplish this:

1. Create an `AgentProperties` instance and pass it to the `Zone.connect` method. The `AgentProperties` constructor requires that the agent's properties be passed to it as an argument. By doing so, the zone properties can inherit default values from the agent.

2. Call the `Zone.getProperties` method to obtain the `AgentProperties` object for that zone, then call its methods to inspect or change properties

The following example shows how to set the messaging mode to Pull for DZ_ZONE while leaving the agent's default messaging mode to Push for all other zones:

```
public void initialize() throws Exception
{
  // Set default agent properties
  AgentProperties defaults = getProperties();
  defaults.setMessagingMode( AgentProperties.PUSH_MODE );

  // Create a couple of zones...
  ZoneFactory factory = getZoneFactory();
  Zone[] zones = new Zone[] {
    factory.getInstance( "COUSINS_MS", "http://localhost:7080/Cousins_MS" ),
    factory.getInstance( "PORTERDALE_MS", 
"http://localhost:7080/Porterdale_MS" ),
    factory.getInstance( "DZ_ZONE", "https://zis.org:7443/DZ_ZONE" ),
  };

  // Change the messaging mode to Pull mode for the DZ_ZONE only
  AgentProperties zoneProps = new AgentProperties( defaults );
  zoneProps.setMessagingMode( AgentProperties.PULL_MODE );
```

Attaching User Data to a Zone

To attach your own arbitrary Object to a *Zone* instance, call the `Zone.setUserData` method. To retrieve the object at a later time, call the `Zone.getUserData` method and cast the result to the appropriate class. Note the ADK will retain a reference to your object for as long as the agent is running, so if the object must be garbage collected be sure to call `setUserData` with a null value when it is no longer used by your agent.

Obtaining the SIF_ZoneStatus Object

Zone Integration Servers report administrative information about zones, such as the agents registered in the zone and the list of providers and subscribers for each SIF Data Object type, in a `SIF_ZoneStatus` object. An agent can query for `SIF_Zon-eStatus` in the same way it queries for other objects—by issuing a `SIF_Request` message and processing the `SIF_Response` returned by the ZIS.

The `Zone.getZoneStatus` method provides a simple and direct way to obtain the `SIF_ZoneStatus` object for a zone. Behind the scenes, this method sends a `SIF_Re-quest` message to the zone integration server and blocks until a response is received. Refer to the GetZoneStatus ADK example agent to see how this is done.

Of course, you may also request `SIF_ZoneStatus` the conventional way by sending a SIF_Request message for it. Simply call the `Zone.query` method and pass a `Query` instance that references the `SIFDTD.SIF_ZONESTATUS` object. The QueryZoneStatus ADK example agent shows this approach in action. Note that by querying for `SIF_ZoneStatus` you must register a *QueryResults* message handler to receive the response asynchronously, whereas the `Zone.getZoneStatus` method does not require a message handler.

Public Interface

Refer to the Javadoc for a complete description of each method.

Method	Description
`connect`	Connect to the zone. SIF Infrastructure messages, including SIF_Register, SIF_Subscribe, SIF_Provide, and SIF_SystemControl / SIF_Ping, may be sent to the zone based on the flags passed to this method and the message handlers that are registered with the *Zone* instance or any *Topic* to which it is joined.
`disconnect`	Disconnect from the zone. The SIF_Unregister message is sent only when the `ADKFlags.PROV_UNREGISTER` flag is specified. No other messages are sent to the Zone Integration Server.
`getAgent`	Gets a reference to the Agent object
`getErrorHandler`	Gets the *UndeliverableMessageHandler* for this zone if one has been previously registered
`getProperties`	Gets the AgentProperties object for this zone. Properties set here are applicable to this zone only. All other properties are inherited from the agent's defaults.
`getUserData`	Retrieves the Object attached to the Zone with the `setUserData` method
`getZoneId`	Gets the Zone ID
`getZoneStatus`	Synchronously obtains the SIF_ZoneStatus object from the Zone Integration Server, or times out if no response is received from the ZIS in the allotted period of time.
`getZoneUrl`	Gets the Zone URL
`isConnected`	Determines if this zone is in a connected state
`isSleeping`	Determines if this zone is sleeping
`publishEvent`	Publishes a SIF_Event to the zone
`purgeQueue`	Purges all messages from the agent's queue
`query`	Issues a SIF_Request query to this zone
`setErrorHandler`	Registers a *UndeliverableMessageHandler* handler with this zone to be notified when messages are received but cannot be dispatched to the ADK's message handlers. Note: Agents do not typically call this method on a zone-by-zone basis but instead install a global error handler by calling `Agent.setErrorHandler`
`setProperties`	Sets the AgentProperties object for this zone
`setPublisher`	Registers a *Publisher* message handler directly with this zone, optionally sending a `SIF_Provide` message to the Zone Integration Server.
`setQueryResults`	Registers a *QueryResults* message handler directly with this zone.
`setReportPublisher`	Registers a *ReportPublisher* message handler directly with this zone, optionally sending a `SIF_Provide` message to the Zone Integration

Method	Description
	Server.
`setSubscriber`	Registers a *Subscriber* message handler directly with this zone, optionally sending a `SIF_Subscribe` message to the Zone Integration Server.
`setUserData`	Attaches an arbitrary Java Object to this *Zone*
`sifPing`	Sends a `SIF_SystemControl` / `SIF_Ping` message to the zone.
`sifProvide`	Sends a `SIF_Provide` message to the zone. Agents do not typically call this method as provisioning is handled by the class framework, but it is provided nonetheless.
`sifRegister`	Sends a `SIF_Register` message to the zone. Agents do not typically call this method as registration is handled by the class framework, but it is provided nonetheless.
`sifSend`	Sends arbitrary XML content to the Zone Integration Server.
`sifSleep`	Sends a `SIF_SystemControl` / `SIF_Sleep` message to the zone
`sifSubscribe`	Sends a `SIF_Subscribe` message to the zone. Agents do not typically call this method as provisioning is handled by the class framework, but it is provided nonetheless.
`sifUnprovide`	Sends a `SIF_Unprovide` message to the zone. This is the only method that sends this message.
`sifUnregister`	Sends a `SIF_Unregister` message to the zone. Agents typically call the `Zone.disconnect` method with the `ADKFlags.PROV_UNREGISTER` flag instead of calling this method directly.
`sifUnsubscribe`	Sends a `SIF_Unsubscribe` message to the zone. This is the only method that sends this message.
`sifWakeup`	Sends a `SIF_SystemControl` / `SIF_Wakeup` message to the zone.
`sleep`	Sends a `SIF_SystemControl` / `SIF_Sleep` message to the zone. There is no difference in calling this method over `sifSleep`
`wakeup`	Sends a `SIF_SystemControl` / `SIF_Wakeup` message to the zone. There is no difference in calling this method over `sifWakeup`

Zones Summary

- The *Zone* interface encapsulates zones in the ADK
- Obtain a *Zone* instance by calling methods of the agent's `ZoneFactory`
- Zones may be joined with Topics (described in the next section)
- Messages may be received by the agent as soon as `Zone.connect` is called
- Prior to connecting a zone, an agent must either register message handlers directly with the *Zone* instance or join the zone with a *Topic*

- Most properties can be customized on a zone-by-zone basis
- To send a `SIF_Register` message when connecting to a zone, pass the `ADKFlags.PROV_REGISTER` flag to the `connect` method. Agents are encouraged to always send a `SIF_Register` message.
- Implement a Connection Watchdog thread to retry failed connection attempts

About Topics

Rather than communicate with zones directly, agents can perform all publish, subscribe, and query activity via *Topics*. A topic is a concept unique to the ADK that aggregates messaging across multiple zones. Use Topics if you want to structure your code to be *data-centric* instead of *zone-centric*. The use of Topics is optional.

Each *Topic* instance is associated with a single type of SIF data object—such as StudentPersonal, LibraryPatronStatus, or BusInfo. When calling the methods of the *Topic* class, the action is applied to all *Zones* "joined" with that topic. When an inbound message is received from a zone, it is dispatched to the *Topic* associated with the data contained in the message. In this way, your agent's message handlers can respond to messages in the same way regardless of which zone the message was received from.

Topic Objects

Topics are represented by the `openadk.library.Topic` interface. During agent startup, you create *Topic* instances for each of the SIF Data Object types your agent will work with by calling the methods of the `TopicFactory` class. The topic factory is a global resource of the agent that serves to create new *Topic* objects and to keep track of which topics the agent has created. Call the `Agent.getTopicFactory` method to obtain a reference to the topic factory:

```
public void initialize() throws Exception
{
  super.initialize();

  // Create Topic instances for each SIF data object type the agent will use
  TopicFactory factory = getTopicFactory();
  ...
```

Next, call the `getInstance` method to obtain a *Topic* instance and pass a `SIFDTD` constant that identifies a SIF Data Object type. If this is the first time calling this method for a topic, the factory returns a new *Topic* object. Subsequently, the same object will be returned.

```
  // Create Topic instances for each SIF data object type the agent will use
  TopicFactory factory = getTopicFactory();
  Topic students = factory.newInstance( SIFDTD.STUDENTPERSONAL );
  Topic staff = factory.newInstance( SIFDTD.STAFFPERSONAL );
```

Registering Message Handlers with Topics

To register message handlers with *Topic* instances, call the following methods:

- `Topic.setPublisher( Publisher handler, int flags )`
- `Topic.setReportPublisher( ReportPublisher handler, int flags )`

- `Topic.setSubscriber( Subscriber handler, int flags )`
- `Topic.setQueryResults( QueryResults handler )`

When the ADK receives an incoming `SIF_Request` message, it is dispatched to your topic's message handler if the payload contains a SIF Data Object for which the topic was established.

The `setPublisher` and `setSubscriber` methods accept two parameters: a reference to the *Publisher* or *Subscriber* message handler you're registering with the topic, and an integer that describes the provisioning options for the registration. One or more constants from the `ADKFlags` class may be specified.

The provisioning options instruct the class framework to send `SIF_Provide` and `SIF_Subscribe` messages, respectively, to each zone that is joined with the topic. To cause `SIF_Provide` to be sent to zones, specify the `ADKFlags.PROV_PROVIDE` flag. Specify the `ADKFlags.PROV_SUBSCRIBE` flag to cause a `SIF_Subscribe` message to be sent to each zone. If you do not want these messages to be sent at all, specify `ADKFlags.PROV_NONE`.

Important: To ensure that SIF_Subscribe and SIF_Provide messages are sent to each zone, be sure to create Topics and register message handlers with them before connecting to zones.

Joining Zones with Topics

In order for your topic to receive SIF messages, you must "join" one or more zones with the topic. You can join the same zone with multiple topics:

```
// Create a Topic instance for StudentPersonal
TopicFactory factory = getTopicFactory();
Topic students = factory.newInstance( SIFDTD.STUDENTPERSONAL );

// Register a Subscriber with the topic so that all SIF_Event messages
// that contain a StudentPersonal payload will be dispatched to this
// topic's handler
Subscriber stuView = (Subscriber)MyDatabase.GetStudentView();
students.setSubscriber( stuView, ADKFlags.PROV_SUBSCRIBE );

// Create Zones
ZoneFactory zoneFact = getZoneFactory();
Zone elemZone = zoneFactory.newInstance( "ELEM",
    "http://10.0.0.4:7080/ELEM" );
Zone dzZone = zoneFactory.newInstance( "DZ", "http://10.0.0.4:7080/DZ" );

// Join the Zones with the Topic
students.join( elemZone );
students.join( dzZone );
```

In the above example, the "ELEM" and "DZ" zones are each joined with the StudentPersonal topic and a *Subscriber* has been registered with it to handle `SIF_Event` messages. Any `SIF_Event` that contains a StudentPersonal payload—regardless of the zone from which it originates—is dispatched to the message handler for that topic. Since *Topics* take precedence over *Zones* in the message dispatching chain, this would be the case even if you registered the *Subscriber* with each of the zones directly.

Public Interface

Refer to the Javadoc for a complete description of each method.

Method	Description
`getObjectType`	Returns a `SIFDTD` constant identifying the SIF Data Object type represented by this topic
`getPublisher`	Returns the *Publisher* message handler registered with this topic to handle `SIF_Request` messages
`getQueryResultsObject`	Returns the *QueryResults* message handler registered with this topic to handle `SIF_Response` messages
`getSubscriber`	Returns the *Subscriber* message handler registered with this topic to handle `SIF_Event` messages
`getZones`	Returns the *Zone* objects joined with this topic
`join`	Joins a *Zone* with this topic
`query`	Issues a `SIF_Request` query to all zones joined with this topic
`setPublisher`	Registers a *Publisher* message handler with this topic to handle `SIF_Request` messages
`setQueryResults`	Registers a *QueryResults* message handler with this topic to handle `SIF_Response` messages
`setReportPublisher`	Registers a *ReportPublisher* message handler with this topic to handle `SIF_Response` messages for SIF_ReportObject objects. Only Vertical Reporting agents that use Topics should call this method.
`setSubscriber`	Registers a *Subscriber* message handler with this topic to handle `SIF_Event` messages

Topics Summary

- A topic represents a SIF Data Object type and a single point of processing for all SIF Messages received by the agent where that object type is contained in the message payload
- Topics offer a *data-oriented* way to structure your code, independent of the zones your agent is connected to
- Topics are optional. Agents can work directly with *Zone* instances
- Topics encapsulated by the `openadk.library.Topic` interface
- Obtain a *Topic* instance by calling methods of the agent's `TopicFactory`
- Call the `setPublisher`, `setSubscriber`, and `setQueryResults` method to register message handlers with the topic before joining zones with it
- Call the `Zone.connect` method after the zone has been joined with topics

7. Publishing

SIF Agents "publish" objects to zones by responding to `SIF_Request` queries. An agent is not required to publish any object types, but for most it makes sense to publish data for which the application is authoritative. A library automation agent, for instance, would be expected to publish the LibraryPatronStatus object since it is the application that houses circulation records for students and staff.

`SIF_Request` messages received by the ADK are dispatched to the agent's *Publisher* message handlers based on the zone the message is received from and the type of object requested in the query conditions. To process `SIF_Requests`, implement the *Publisher* interface and register it with one or more Topic or Zone objects as described in Working with SIF Messages on page 32.

Agents typically respond to `SIF_Request` messages by following these steps:

1. Evaluate the query conditions to select records from the application's database

2. If the query is not supported by your agent, prepare and throw a `SIFException` with an error category of `SIFErrorCodes.CAT_REQRSP_8` and an error code of `SIFErrorCodes.REQRSP_UNSUPPORTED_QUERY_9`. (NOTE: The ADK will automatically return a similar error acknowledgement when a `SIF_Request` is received for an object type that's not supported by the agent, so you don't need to check this condition in your own code.)

3. Query the application's database or other data source for records matching the query conditions. Convert the resulting records to `SIFDataObject` instances, either programmatically or using classes such as `Mappings`

4. Stream the set of `SIFDataObjects` to the `DataObjectOutputStream` supplied by the class framework

All of the underlying tasks associated with creating and sending `SIF_Response` messages are handled by the framework. This includes dividing the response stream into one or more packets based on the buffer size of the requesting agent; removing unnecessary elements from objects per the requestor's instructions in the `SIF_QueryObject/SIF_Element` of the request; rendering SIF Data Objects in the version of SIF requested in the `SIF_Request/SIF_Version` element; and so on.

When you stream objects to the `DataObjectOutputStream`, the ADK writes them to the agent's work directory on the file system. They are then converted to `SIF_Response` message and delivered to the zone integration server immediately. If there is a failure – for example, the agent stops running or the computer is powered down – the class framework picks up where it left off the next time the agent is run.

The Publisher Interface

The *Publisher* interface is implemented by classes that can respond to `SIF_Requests`. It has one method:

```
public void onRequest(
  DataObjectOutputStream results, Query query, Zone zone, MessageInfo info )
    throws ADKException
```

The goal of the `Publisher.onRequest` method is to evaluate the supplied `Query`, create one or more `SIFDataObjects` that meet the query conditions, and then stream those data objects to the supplied `DataObjectOutputStream`.

You can stream an arbitrarily large amount of data. The ADK automatically divides the results into packets in order to send one or more `SIF_Response` messages to the zone. It also takes care of rendering the `SIFDataObjects` using the version of SIF indicated in the SIF_Request message and trims objects to size based on the SIF_Element restrictions in the query.

The DataObjectOutputStream Parameter

Returning data from the onRequest method is very straightforward: simply create one or more SIFDataObjects and call the DataObjectOutputStream.write method for each:

```
public void onRequest(
  DataObjectOutputStream results, Query query, Zone zone, MessageInfo info )
    throws ADKException
{
  // Return a bogus student object
  StudentPersonal student = new StudentPersonal();
  student.setRefId( "E3E34B359D75101A88C3D00AA00184754" );
  student.setName( new Name( NameType.BIRTH, "Alicia", "Johnson" ) );

  // Write to the output stream
  results.write( student );
}
```

Sending Raw XML

You can also send raw XML content by using the SIFDataObjectXML class. Keep in mind, however, that because you are preparing your own objects as XML instead of the class framework, it will not be able to perform two important functions that are normally performed automatically:

- No multiple version support
- No automatic trimming of SIF elements to match the query's field restrictions

To send raw XML content:

```
public void onRequest(
  DataObjectOutputStream results, Query query, Zone zone, MessageInfo info )
    throws ADKException
{
  // Create a <StudentPersonal> object
  String xml =
    "<StudentPersonal RefId='E3E34B359D75101A88C3D00AA00184754'>" +
      "<Name Type='01'>" +
        "<LastName>Johnson</LastName" +
        "<FirstName>Alicia</FirstName>" +
      "</Name>" +
    "</StudentPersonal>";

  // Wrap the string in a SIFDataObjectXML object
  SIFDataObjectXML student = new SIFDataObjectXML( SIFDTD.STUDENTPERSONAL,
xml );

  // Write to the output stream
  results.write( student );
}
```

The Query Parameter

The Query parameter to onRequest describes the SIF object or objects to return in the response. In SIF 1.0 there are two basic flavors of request: those with query conditions and those without. Call `Query.hasConditions` to quickly determine if a query has any conditions:

```
public void onRequest(
  DataObjectOutputStream results, Query query, Zone zone, MessageInfo info )
    throws ADKException
{
  if( query.hasConditions() )
  {
    // Evaluate the query
  }
  else
  {
    // Return all objects
  }
```

Evaluating Conditions

To enumerate all conditions of a query, call the `getConditions` method. Conditions are organized by `ConditionGroup`, each of which is comprised of one or more `Condition` objects. In SIF 1.0 there is only one root `ConditionGroup`, but in future versions of SIF this will be expanded.

The `ConditionGroup` and `Condition` classes model the `SIF_ConditionGroup` and `SIF_Condition` elements described by the SIF Specification.

Because of the limited query support offered by SIF 1.0, agents typically request all objects or a single object given a *RefId* value. The `Query.hasCondition` method can be used to quickly determine if a specific, well-known condition has been placed on the query. The following code snippets illustrate how to use the `hasCondition` method to respond to a query without implementing comprehensive query analysis logic:

```
public void onRequest(
  DataObjectOutputStream results, Query query, Zone zone, MessageInfo info )
    throws ADKException
{
  // Either query for all StudentPersonal objects or for a single
  // object by RefId
  Condition c = query.hasCondition( SIFDTD.STUDENTPERSONAL_REFID );
  if( c != null )
  {
    // Lookup the student with the specified RefId
    String refId = c.getValue();

    ...
  }
  else
  {
    // Query for all students...
  }
```

This example involves `LibraryPatronStatus` objects:

```
public void onRequest(
  DataObjectOutputStream results, Query query, Zone zone, MessageInfo info )
    throws ADKException
{
  // Either query for all LibraryPatronStatus objects or for a single object
  // for the Student or Staff identified by RefId

  Condition c = query.hasCondition( SIFDTD.LIBRARYPATRONSTATUS_SIFREFID );
  if( c != null )
  {
    // Get the RefId value
    String refId = c.getValue();

    // Student or staff?
    c = query.hasCondition( SIFDTD.LIBRARYPATRONSTATUS_SIFREFIDTYPE );
    if( c.getValue().equals( "StudentPersonal" ) )
    {
      // Query for the student identified by RefId
    }
    else
    if( c.getValue().equals( "StaffPersonal") )
    {
      // Query for the staff identified by RefId
    }
    else
      throw new SIFException(
        SIFErrorCodes.CAT_XML_1,
        SIFErrorCodes.XML_INVALID_VALUE_4,
        "SifRefIdType must specify StudentPersonal or StaffPersonal",
        c.getValue() + " is not a valid value for this attribute" );
  }
  else
  {
    // Query for all LibraryPatronStatus objects...
  }
```

Object Type

If your *Publisher* interface is registered with a Topic, your implementation will inherently know the type of object being queried for. This is because you can only register one *Publisher* with a given topic, and topics are always associated with one and only one object type. So when the *Publisher* message handler is called on your `StudentPersonal` topic, you can assume the request is for `StudentPersonal` objects.

However, if you are not using topics—or more commonly, are reusing the same *Publisher* implementation among many topics—the object type associated with the request can be obtained from the `Query.getObjectType` method:

```
public void onRequest(
  DataObjectOutputStream results, Query query, Zone zone, MessageInfo info )
    throws ADKException
{
  if( query.getObjectType() == SIFDTD.STUDENTPERSONAL )
  {
    // Query for StudentPersonal objects...
  }
  else
  if( query.getObjectType() == SIFDTD.SCHOOLINFO )
  {
    // Query for SchoolInfo objects...
  }
```

Notice the `getObjectType` method returns a `SIFDTD` constant to uniquely identify the type of object in a version-independent way. This approach is used consistently throughout the ADK because the string names of elements and attributes in SIF may change from version to version. By using `SIFDTD` constants your code is portable from one version of SIF to the next.

ⓘ If you're not familiar with the SIFDTD class, please refer to Creating & Manipulating SIFDataObjects on page 20.

The ADK supplies two ways to create and manipulate `SIFDataObject` instances. The first is to programmatically construct objects and call the methods of the SDO classes to get and set element and attribute values. For example,

```
// Constructing a StudentPersonal object
StudentPersonal sp = new StudentPersonal();
sp.setRefId( refid );
sp.setName( NameType.LEGAL, "Johnson", "Clifford" );

// Examining a StudentPersonal object
Name n = sp.getName();
System.out.println( "Name: " + n.getLastName() + ", " + n.getFirstName() );
OtherId[] ids = sp.getOtherIds();
System.out.println( "This student has " + ids.length + " IDs" );
```

You can also use the setElementOrAttribute method:

```
// Dynamically constructing a StudentPersonal object
SIFDataObject sp = ADK.DTD().createSIFDataObject( SIFDTD.STUDENTPERSONAL );
sp.setElementOrAttribute( "@RefId", refid );
sp.setElementOrAttribute( "Name[@Type='01']/LastName", "Johnson" );
sp.setElementOrAttribute( "Name[@Type='01']/FirstName", "Clifford" );

// Dynamically examining a StudentPersonal object
String refId = sp.getElementOrAttribute( "@RefId" );
String studentId = sp.getElementOrAttribute( "OtherId[@Type='06'" );
```

An alternative and more data-driven approach to working with `SIFDataObjects` is to use the `Mappings` class from the `openadk.library.tools.Mappings` package to convert `SIFDataObject` instances to and from a `HashMap` by applying a set of XPath-like mapping rules. These rules are usually stored in an external configuration file where they can be modified by system integrators in the field. For example, the following rules for StudentPersonal objects define how to translate the elements and attributes of that object to a flat list of field values:

```
<object object="StudentPersonal">
       <field name="STUDENT_ID">OtherId[@Type='01']</field>
       <field name="LAST_NAME">Name[@Type='01']/LastName</field>
```

```
        <field name="FIRST_NAME">Name[@Type='01']/FirstName</field>
</object>
```

You can then call upon the `Mappings` class to convert a StudentPersonal instance into a `HashMap` of field/value pairs, or to convert a `HashMap` into a StudentPersonal object.

In practice, most agents use a combination of these two approaches when working with SIF Data Objects. Refer to the chapter on SIF Data Objects for more information

Sometimes it is necessary to use the `SIFDTD` constant as a key into your own lookup tables, or to otherwise obtain the string name of an element or attribute. To obtain the version-*independent* name of a `SIFDTD` constant, call its `name()` method. For the version-*dependent* name—which is specific to the version of SIF associated with the message being processed and may change with each version of SIF—call the `getObjectTag` convenience method.

The following code calls the `name` method as a way of determining the name of a table in the application's SQL database, and uses the `getObjectTag` method to print a message to the Java console:

```
public void onRequest(
  DataObjectOutputStream results, Query query, Zone zone, MessageInfo info )
    throws ADKException
{
  SIFMessageInfo inf = (SIFMessageInfo)info;

  // In our database, tables are named the same as SIF 1.0r1 objects:
  String sql = "SELECT * FROM " + query.getObjectType().name();

  // Print the version-dependent tag name of the object we're querying for
  System.out.println(
    "Agent " + inf.getSourceId() + " in zone " + zone.getId() + " is " +
    "querying for all " + query.getObjectTag() + " objects"

  // Execute the query
  ...
```

Using the QueryFormatter Classes

For agents that need to prepare a query string for SQL or another form of query language, the `openadk.library.tools.QueryFormatter` utility class offers a simple solution. This class builds a string from a dictionary of field names you supply. The text to use for logical `AND` and `OR` operators is configurable, and by subclassing `QueryFormatter` you can change the way fields are rendered to the query string.

An example of how to subclass `QueryFormatter` is found in the `SQLQueryFormatter` class, which causes string values to be surrounded with single quotes, the text "AND" to be used for logical AND operations, and the text "OR" to be used for logical OR operations. Source code is provided in the `\extras` directory.

Mapping SIF Attributes to Local Field Names

In order for the basic `QueryFormatter` class to work, you must supply it with a `Dictionary` that maps SIF attributes to field names of your choosing. The `format` function uses these field names in place of SIF attributes specified in the query condition. When building the `Dictionary`, the key values must be constants from the `SIFDTD` class.

Mapping SIF Values

If the value of a mapping contains curly braces, the text within the braces maps field *values.* The format of this string must be:

```
"field-name{value1|str1|value2|str2|…}"
```

where value*N* is replaced with str*N*.

This feature is handy if you need to substitute a SIF attribute value with a value specific to your application. For example, the acceptable values for the `SifRefIdType` attribute of the `LibraryPatronStatus` object is “StudentPersonal” or “StaffPersonal”. If in your application’s database you represented these as numeric types—say 1 and 2, respectively—you could create a field mapping as follows:

```
"CircStatus.Type{StudentPersonal|1|StaffPersonal|2}"
```

Another way to use this feature is to substitute *any* value with the specified text. This can be done by simply leaving the *valueN* component blank. In the following example, the value for the SIF attribute mapped to the “Transportation.Direction” field will always be the hard-coded value 16, regardless of the actual value specified in the query conditions:

```
"Transportation.Direction{|16}"
```

Of course, you can subclass QueryFormatter to modify the default behavior or to add your own functionality, but by default it performs mappings as described herein.

An Example

This code shows how to use the `SQLQueryFormatter` class:

```
  // Static dictionary that maps SIF Data Object names to tables in our
database schema
  static HashMap sMyTables = new Dictionary();
  static {
    sMyTables.put( SIFDTD.STUDENTPERSONAL.name(), "Students" );
    sMyTables.put( SIFDTD.STAFFPERSONAL.name(), "Staff" );
    sMyTables.put( SIFDTD.LIBRARYPATRONSTATUS.name(), "CircStatus" );
  };

  // Static dictionary that maps SIFDTD constants to field names in our
database schema
  static HashMap sMyFields = new Dictionary();
  static {
    sMyFields.put( SIFDTD.STUDENTPERSONAL_REFID, "Students.SifGUID" );
    sMyFields.put( SIFDTD.STAFFPERSONAL_REFID, "Teachers.SifGUID" );
    sMyFields.put( SIFDTD.LIBRARYPATRONSTATUS_SIFREFID, "CircStatus.SifGUID" );
    sMyFields.put( SIFDTD.LIBRARYPATRONSTATUS_SIFREFIDTYPE,
      "CircStatus.Type{StudentPersonal|1|StaffPersonal|2|}" );
    sMyFields.put( SIFDTD.LIBRARYPATRONSTATUS_LIBRARYTYPE, "{|
  };

  // Static QueryFormatter
  static QueryFormatter sFormatter = new SQLQueryFormatter();

  public void onRequest(
    DataObjectOutputStream results, Query query, Zone zone, MessageInfo info )
      throws ADKException
  {
    String table = query.getObjectType().name();

    // Build an SQL statement
    String sql = "SELECT * FROM " + table + " WHERE " +
      sFormtter.format( query, sMyFields );
```

If the above code received a `SIF_Request` for `LibraryPatronStatus`:

```
<SIF_Request>
   …
   <SIF_Query>
      <SIF_QueryObject ObjectName="LibraryPatronStatus">
      <SIF_ConditionGroup Type="And">
         <SIF_Conditions Type="Or">
            <SIF_Condition>
               <SIF_Element>@SifRefId</SIF_Element>
               <SIF_Operator>EQ</SIF_Operator>
               <SIF_Value>xxx</SIF_Value>
            </SIF_Condition>
            <SIF_Condition>
               <SIF_Element>@SifRefId</SIF_Element>
               <SIF_Operator>EQ</SIF_Operator>
               <SIF_Value>yyy</SIF_Value>
            </SIF_Condition>
            <SIF_Condition>
               <SIF_Element>@SifRefId</SIF_Element>
               <SIF_Operator>EQ</SIF_Operator>
               <SIF_Value>zzz</SIF_Value>
            </SIF_Condition>
         </SIF_Conditions>
         <SIF_Conditions Type="None">
            <SIF_Condition>
               <SIF_Element>@SifRefIdType</SIF_Element>
               <SIF_Operator>EQ</SIF_Operator>
               <SIF_Value>StudentPersonal</SIF_Value>
            </SIF_Condition>
         </SIF_Conditions>
      </SIF_ConditionGroup>
   </SIF_Query>
</SIF_Request>
```

The result of calling `SQLQueryFormatter.format` would be:

```
SELECT * FROM CircStatus WHERE ( CircStatus.SifGUID = 'xxx' OR Circ-
Status.SifGUID = 'yyy' OR CircStatus.SifGUID = 'zzz' ) AND ( Circ-
Status.Type = 1 )
```

The Zone Parameter

The third parameter to `Publisher.onQuery` is the *Zone* object identifying the zone from which the `SIF_Request` message originated.

The zone is an important piece of information when processing requests because it defines the scope of records to return from the application's database. A request for "all StudentPersonal objects" should be interpreted as "all StudentPersonal objects from this zone". Although the SIF Specification places no restrictions on the organization of zones in a district, Data Solutions recommends that each school be represented by its own zone, and an additional zone be created for the district itself. In this scenario, you could use the *Zone* object to select records from the appropriate database by maintaining a mapping of schools-to-zones.

The SIFMessageInfo Parameter

The final parameter to `Publisher.onQuery` is a `MessageInfo` object, which provides miscellaneous information about the message such as its header fields, message ID, and the *SourceId* of the sender. For SIF_Request messages, additional fields such as `SIF_Request/SIF_MaxBufferSize` and `SIF_Request/SIF_Version` are available. These attributes can be retrieved by casting the object to a `SIFMessageInfo` and calling the following methods:

- `SIFMessageInfo.getSourceId`
- `SIFMessageInfo.getMsgId`
- `SIFMessageInfo.getSIFRequestVersion`
- `SIFMessageInfo.getSIFMaxBufferSize`

```
  public void onRequest(
    DataObjectOutputStream results, Query query, Zone zone, MessageInfo info )
      throws ADKException
  {
    SIFMessageInfo inf = (SIFMessageInfo)info;

    System.out.println( "Received a SIF_Request from " + inf.getSourceId() +
      " in zone " + zone.getZoneId() );
    System.out.println( "The SIF_MaxBufferSize value is: " +
inf.getSIFMaxBufferSize() );
    System.out.println( "The SIF_Version value is: " +
inf.getSIFRequestVersion() );
    System.out.println( "The version of the SIF_Request message is: " +
      inf.getSIFVersion() );
```

Refer to the Javadoc for more information on the `SIFMessageInfo` class.

8. Subscribing

The Subscriber Interface

The *Subscriber* interface is implemented by classes that can respond to `SIF_Events`. It has one method:

```
public void onEvent( Event event, Zone zone, MessageInfo info )
  throws ADKException
```

Event Objects

The `Event` object provides the SIF Data Objects communicated by the event and describes what action should be taken on those objects. Call the `getAction` method to determine if the objects were added, changed, or deleted. This method returns one of three constants defined by the `Event` class:

- `Event.ADD`
- `Event.CHANGE`
- `Event.DELETE`

To retrieve the actual `SIFDataObject`s, call the `getData` method and then read from the returned `DataObjectInputStream` until no more objects are available:

```
public void onEvent( Event event, Zone zone, MessageInfo info )
  throws ADKException
{
  DataObjectInputStream data = event.getData();

  while( data.available() )
  {
    StudentPersonal student = (StudentPersonal)data.readDataObject();

    switch( event.getAction() )
    {
      case Event.ADD:
        // Add this student to the local database

      case Event.CHANGE:
        // Update this student in the local database

      case Event.DELETE:
        // Remove this student from the local database

    }
  }
}
```

Object Type

If your *Subscriber* interface is registered with a Topic, your implementation will inherently know the type of object communicated by the event. This is because you can only register one *Subscriber* with a given topic, and topics are always associated with one and only one object type. So when the *Subscriber* message handler is called on your StudentPersonal topic, you can assume the event is for StudentPersonal objects.

However, if you are not using topics—or more commonly, are reusing the same *Subscriber* implementation among many topics—the object type associated with the event can be obtained from the Event.getObjectType method:

```
public void onEvent( Event event, Zone zone, MessageInfo info )
  throws ADKException
{
  if( event.getObjectType() == SIFDTD.STUDENTPERSONAL )
  {
    // This Event contains StudentPersonal objects...
  }
  else
  if( event.getObjectType() == SIFDTD.SCHOOLINFO )
  {
    // This Event contains SchoolInfo objects...
  }
```

Notice the getObjectType method returns a SIFDTD constant to uniquely identify the type of object in a version-independent way. This approach is used consistently throughout the ADK because the string names of elements and attributes in SIF may change from version to version. By using SIFDTD constants your code is portable from one version of SIF to the next.

ⓘ If you're not familiar with the SIFDTD class, please refer to Creating & Manipulating SIFDataObjects on page 20.

The ADK supplies two ways to create and manipulate SIFDataObject instances. The first is to programmatically construct objects and call the methods of the SDO classes to get and set element and attribute values. For example,

```
// Constructing a StudentPersonal object
StudentPersonal sp = new StudentPersonal();
sp.setRefId( refid );
sp.setName( NameType.LEGAL, "Johnson", "Clifford" );

// Examining a StudentPersonal object
Name n = sp.getName();
System.out.println( "Name: " + n.getLastName() + ", " + n.getFirstName() );
OtherId[] ids = sp.getOtherIds();
System.out.println( "This student has " + ids.length + " IDs" );
```

You can also use the setElementOrAttribute method:

```
// Dynamically constructing a StudentPersonal object
SIFDataObject sp = ADK.DTD().createSIFDataObject( SIFDTD.STUDENTPERSONAL );
sp.setElementOrAttribute( "@RefId", refid );
sp.setElementOrAttribute( "Name[@Type='01']/LastName", "Johnson" );
sp.setElementOrAttribute( "Name[@Type='01']/FirstName", "Clifford" );

// Dynamically examining a StudentPersonal object
String refId = sp.getElementOrAttribute( "@RefId" );
String studentId = sp.getElementOrAttribute( "OtherId[@Type='06'" );
```

An alternative and more data-driven approach to working with SIFDataObjects is to use the Mappings class from the openadk.library.tools.Mappings package to convert SIFDataObject instances to and from a HashMap by applying a set of XPath-like mapping rules. These rules are usually stored in an external configuration file where they can be modified by system integrators in the field. For example, the following rules for StudentPersonal objects define how to translate the elements and attributes of that object to a flat list of field values:

```
<object object="StudentPersonal">
	<field name="STUDENT_ID">OtherId[@Type='01']</field>
	<field name="LAST_NAME">Name[@Type='01']/LastName</field>
	<field name="FIRST_NAME">Name[@Type='01']/FirstName</field>
</object>
```

You can then call upon the `Mappings` class to convert a StudentPersonal instance into a `HashMap` of field/value pairs, or to convert a `HashMap` into a StudentPersonal object.

In practice, most agents use a combination of these two approaches when working with SIF Data Objects. Refer to the chapter on SIF Data Objects for more information

The Zone Parameter

The second parameter to `Subscriber.onEvent` is the *Zone* object identifying the zone from which the `SIF_Event` message originated. You can use this information to select the appropriate database to update with the objects contained in the event. For example, if a new student is added, your agent can consult its schools-to-zones table to determine which school the student should be added to.

The SIFMessageInfo Parameter

The final parameter to `Subscriber.onEvent` is a `MessageInfo` object, which provides miscellaneous information about the message such as its header fields, message ID, and the *SourceId* of the sender. These attributes can be retrieved by casting the object to a `SIFMessageInfo`.

```
  public void onRequest(
    DataObjectOutputStream results, Query query, Zone zone, MessageInfo info )
      throws ADKException
  {
    SIFMessageInfo inf = (SIFMessageInfo)info;

    // In order to get the verion-specific tag name of the SIF Data Object, the
    // version of SIF associated with the SIF_Event must be obtained from the
MessageInfo
    String objectTag = event.getObjectType().tag( inf.getSIFVersion() );

    // Write a message to the console (e.g. "Received a SIF_Event::Add for
StudentPersonal
    // from agent MyAgent in zone MyZone")
    System.out.println(
      "Received a SIF_Event::" + event.getActionStr() +
      " for " + objectTag +
      " from agent " + inf.getSourceId() +
      " in zone " + zone.getZoneId() );
```

Refer to the Javadoc for more information on the `SIFMessageInfo` class.

Executing Queries from Subscriber.onEvent

If the processing of a `SIF_Event` will require that your agent query for additional data in order to complete the operation, the `TrackQueryResults` class must be used to perform the query synchronously. This is required so that Selective Message Blocking will be enabled if needed.

Selective Message Blocking and the ADK

Selective Message Blocking is a feature of SIF that prevents deadlock conditions when requesting objects during SIF Event processing. When invoked, it marks all SIF Events as "frozen" in the agent queue on the server so that subsequent SIF Response messages will be returned to the agent. Once all SIF Responses have been received and the agent is finished processing the SIF Event, it removes the Selective Message Blocking mode from its queue.

Selective Message Blocking is handled automatically by the ADK's `TrackQueryResults` class. When using this class from within your *Subscriber* message handler, pass the handler's `Event` parameter to the `TrackQueryResults` constructor. `TrackQueryResults` will then invoke SMB on that SIF Event and will subsequently clear the block when you're finished using it. You do not need to worry about sending the appropriate `SIF_Ack` messages to invoke and clear Selective Message Blocking and in fact the ADK does not provide a mechanism to do so manually.

9. Event Reporting

SIF Agents that publish objects to a zone are responsible for letting other agents know when those objects have changed or been deleted, as well as when new objects have been added. Similarly, agents that wish to communicate changes in shared SIF objects should also publish events so that other applications will be notified the data has changed. For example, if a student's address was corrected in the Library application, it should report the change so the Student Information System can update its authoritative copy of the student data.

When an event is reported by your agent, the Zone Integration Server forwards it on to all subscribers of that object type. System administrators often configure the ZIS to disallow event reporting by some agents as a means of enforcing policy. For instance, if a district requires that changes to student information can *only* be made in the Student Information System, the administrator can disallow event reporting by all other agents in the zone.

When to Report Events

Agents should report SIF Events when a shared SIF object is added, changed, or deleted. However, we recommend that agents make event reporting a configurable option so that system administrators can fine-tune the agent to cut down on network traffic. This way, if an administrator knows that event reporting will be disallowed in an agent's Access Control List, he can also disable it in the agent's configuration file.

Reporting Events to the Appropriate Zone

You should take care to write your agent to report events only to the zone associated with the data objects that have been added, changed, or deleted by your application. This is particularly important when reporting added objects. For example, if a new student is added to East High School, your agent should only report an event to the zone associated with that school. Reporting the change to all zones would result in the new student being added to each of the other schools' databases, which is probably not the desired effect. For this reason, the *Topic* interface does not have a `reportEvent` method.

How to Report Events

Event reporting in the ADK is performed by calling the `reportEvent` method on a *Zone* instance. There are two forms of this method: one that accepts a SIFDataObject and a constant identifying how the object has changed (i.e. added, changed, or deleted); and another that accepts this same information encapsulated by an `Event` instance:

```
public void reportEvent( SIFDataObject object, byte actionCode )
  throws ADKException;

public void reportEvent( Event event )
  throws ADKException;
```

Action codes are defined by the Event class:

- `Event.ADD`
- `Event.CHANGE`
- `Event.DELETE`

To report an event for a single object:

```
// Report a new student
StudentPersonal student = ...
myZone.reportEvent( student, Event.ADD );
```

Per SIF Specifications, an agent can only send one `SIFDataObject` per event. However, in the future this restriction may be relaxed so the `Event` class uses arrays and Java streams to ensure it can support an arbitrarily large number of objects in the future.

To wrap an event in an Event object:

```
// Report a new student using the Event object
StudentPersonal student = ...
Event ev = new Event( new SIFDataObject[] { student }, Event.ADD );
myZone.reportEvent( ev );
```

Note the `Event` class is also used by the *Subscriber* message handler when a `SIF_Event` message is received by your agent, so some of the constructors and public methods of the class do not apply to event reporting.

Strategies for Event Reporting

There are many strategies for keeping track of changes made by your application. Some agents report changes immediately at the time they're made by a user in the application's user interface, while others keep track of changes in a "journal" and periodically enumerate the journal to report events to SIF.

Good software engineering practice prescribes that changes be tracked in your application's data layer—as opposed to in a client layer—to ensure that SIF Events are reported whenever changes are made regardless of how they're made. When arriving at a strategy for SIF Event reporting, consider all of the ways in which data is changed by your application. Chances are SIF should be an integral part of all of these processes.

- Changes made directly in your application's user interface
- Changes made by Import functions
- Changes made by external tools such as an End-of-Year Rollover tool
- Backup and Restore
- *etc.*

Another important aspect of Event Reporting involves reliability. Agents should track and report changes even if those changes occur when the SIF Agent is not running. If you use the ADK to develop an "external agent"—that is, an agent that runs alongside your application—there is always the possibility that the agent will not be running at the same time your application is.

Journaling

One common approach to tracking changes for the purposes of SIF Event reporting is to use a "journal". A journal is a database table or other data store that captures what kind of data changed, how it changed, and when it changed. Your agent can periodically enumerate entries in the journal to generate SIF Events. Once a SIF Event is successfully sent for an entry in the journal, it is removed.

Preparing SIFDataObjects for Event Reporting

When reporting an event, agents are responsible for including only the data that has actually changed. This is important because it allows other agents to quickly determine the scope of the change when updating data in the application's local database. For example, if a student's phone number has changed, your agent should only include the `<PhoneNumber>` element in the StudentPersonal object communicated by the event; all other elements of the StudentPersonal object should be excluded.

There are two schools of thought for accomplishing this with the ADK:

- Create `SIFDataObjects` from scratch from the data in your database

- Use the `SIFElement.setChanged` method to mark elements as "changed"

The method you choose depends on how you've written your agent to interact with your application's database layer.

Method 1: Create SIFDataObjects from Scratch

This approach involves simply creating a new SIFDataObject instance and then assigning values to only those fields that have changed. For example, if a user has just modified the home phone number in your application's user interface, you can create a new StudentPersonal object on-the-fly for the purposes of event reporting. By calling the `StudentPersonal.setPhoneNumber` method, the phone number will be the only element included in the SIF Event.

```
String homePhone = getFieldFromDialogBox( “HomePhone” );

// Create a StudentPersonal to report this change to SIF
StudentPersonal sp = new StudentPersonal();
sp.setRefId( ... );
sp.setPhoneNumber( PhoneNumberType.HOME_PHONE, homePhone );

// Report the event to the zone
myZone.reportEvent( sp, Event.CHANGE );
```

Don’t forget to assign the student object’s RefId value by calling `setRefId`.

Method 2: Use SIFElement.setChanged

The above method works well when creating new `SIFDataObjects` from scratch, but what if you already have an instance in memory and want to use it to report a change?

All SIF Data Object classes derived from `openadk.library.Element` have an internal flag to track state. You can set this flag by calling the `SIFElement.setChanged` method, which marks the element and all of its children as changed. When reporting SIF Events, the ADK only includes changed elements in the event.

For performance reasons, the “changed” flag is only modified when you explicitly call the `setChanged` method rather than automatically whenever you set the value of an element. You should call `setChanged( false )` on the root `SIFDataObject` before marking any child elements as changed to ensure that the object is in a consistent state. This technique is illustrated below.

```
// Assume this StudentPersonal is fully populated
StudentPersonal student = ...

// Reset all elements of StudentPersonal to the “not changed” state
student.setChanged( false );

// Mark the home phone and middle name as changed
student.getPhoneNumber( PhoneNumberType.HOME_PHONE ).setChanged( true );
student.getName().getMiddleName().setChanged( true );

// Now report the event; only <PhoneNumber> and <MiddleName> will
// be included in the StudentPersonal object rendered by the ADK
myZone.reportEvent( student, Event.CHANGE );
```

10. Querying

Agents that consume data from a SIF Zone do so by sending a `SIF_Request` message to query for all SIF Data Objects of a given type, or for only a select subset of objects that match conditions placed on the query. For instance, an agent can query for “all StudentPersonal” objects or for “the StudentPersonal object with RefId xxx”.

When the zone integration server receives a `SIF_Request`, it forwards the message to the agent that is registered as the authoritative Provider of the object type in the zone. Queries may also be directed at a specific agent—in this case the server places the `SIF_Request` message in that agent’s queue only. When the responding agent processes the request, it creates one or more `SIF_Response` messages consisting of the objects requested. The server delivers these messages to the requesting agent's queue. Ultimately, they're received by the ADK and dispatched to the *QueryResults* message handler for the associated SIF Data Object type.

Querying for Data

Querying for SIF Data Objects with the ADK is easy:

- Prepare a `openadk.library.Query` instance to describe the type of object your agent is querying and any conditions to be placed on the query.
- Pass the `Query` instance to the query method of a zone or topic. If the `Zone.query` method is called, a `SIF_Request` message is sent to that zone; if the `Topic.query` method is called, a message is broadcast to all zones joined with the topic.
- To direct the query at a specific agent, pass the *SourceId* of the destination agent to the `query` method.
- Optionally, you can associate a small piece of state information with the query by calling `Query.setUserData`. This state information will be persisted by the ADK and available later on when the response messages are received.

To receive the results of a query,

- Implement the *QueryResults* message handler interface on a class of your choosing and register that handler with a zone or topic by calling the `Zone.setQueryResults` or `Topic.setQueryResults` methods.
- You can provide a single *QueryResults* implementation to handle all `SIF_Response` messages received by the agent, or you can register multiple implements, one for each kind of SIF Data Object your agent queries. When multiple *QueryResults* handlers are registered with a zone or topic, the ADK dispatches incoming `SIF_Response` messages to the appropriate handler by examining the message payload and choosing an implementation based on the type of SIF Data Object contained in the message.

Querying for All Objects

The following example shows how to query a zone for all StudentPersonal objects. No conditions are placed on the `Query`:

```
// Prepare a Query for all StudentPersonal objects
Query q = new Query( SIFDTD.STUDENTPERSONAL );

// Send the query to a zone
Zone myZone = getZoneFactory().getZone( “MYZONE” );
myZone.query( q );
```

To query all zones joined with a topic, call the `Topic.query` method:

```
// Prepare a Query for all StudentPersonal objects
Query q = new Query( SIFDTD.STUDENTPERSONAL );

// Send the query to all zones joined with the StudentPersonal topic
Topic students = getTopicFactory().getInstance( SIFDTD.STUDENTPERSONAL );
students.query( q );
```

Querying for a Specific Object by RefId

The following example shows how to query a zone for a specific StudentPersonal object by specifying a condition on the query.

```
  // Prepare a Query for StudentPersonal objects
  Query q = new Query( SIFDTD.STUDENTPERSONAL );

  // Add a condition to query for a specific RefId
  q.addCondition( SIFDTD.STUDENTPERSONAL_REFID, Condition.EQ,
"4A37969803F0D00322AF0EB969038483");

  // Send the query to a zone
  Zone myZone = getZoneFactory().getZone( "MYZONE" );
  myZone.query( q );
```

The first argument to the `Query.addCondition` method is an `ElementDef` constant from the `SIFDTD` class that identifies the *RefId* attribute of the StudentPersonal object. The second argument is an operator from the Condition class, and the third parameter is the string value to compare against. Taken together, these three parameters place a condition on the query instructing the responding agent to "Return the StudentPersonal object where the *RefId* attribute is equal to 4A37969803F0D00322AF0EB969038483".

Note that the current version of the SIF Specification only supports the equality operator, `Condition.EQ`. The ADK defines additional operators for greater-than and less-than comparisions—`Condition.LT` and `Condition.GT`. Although these operators may be supported by SIF in the future, using them with SIF 1.x will result in an invalid `SIF_Request` message.

Multiple conditions can be placed on a query by calling `Query.addCondition` repeatedly:

```
  // Prepare a Query for LibraryPatronStatus objects
  Query q = new Query( SIFDTD.LIBRARYPATRONSTATUS );

  // Add conditions to query for a specific StaffPersonal RefId
  q.addCondition( SIFDTD.LIBRARYPATRONSTATUS_SIFREFIDTYPE, Condition.EQ,
"StaffPersonal");
  q.addCondition( SIFDTD.LIBRARYPATRONSTATUS_SIFREFID, Condition.EQ,
"4A37969803F0D00322AF0EB969038483");

  // Send the query to a zone
  Zone myZone = getZoneFactory().getZone( "MYZONE" );
  myZone.query( q );
```

Directed Queries

A "directed query" is a SIF_Request message sent to a specific agent in a zone. To send a directed query, use the form of the `Zone.query` or `Topic.query` method that accepts a *DestinationId* parameter.

```
  // Prepare a Query for StudentPersonal objects
  Query q = new Query( SIFDTD.STUDENTPERSONAL );

  // Add a condition to query for a specific RefId
  q.addCondition( SIFDTD.STUDENTPERSONAL_REFID, Condition.EQ,
"4A37969803F0D00322AF0EB969038483");

  // Send the query to the "SASIxp" agent in this zone
  Zone myZone = getZoneFactory().getZone( "MYZONE" );
  myZone.query( q, "SASIxp", 0 );
```

Note the third parameter to the `query` method is a "query flag" reserved for future use. Specify a value of zero.

Placing Field Restrictions on a Query

The SIF query mechanism provides a way for agents to declare the specific elements that should be returned with `SIF_Response` messages. By doing so, you can minimize the amount of data sent over the network. For example, if your agent is only interested in the first and last name of each StudentPersonal object it receives, adding "field restrictions" to the query will instruct the recipient to return only these two elements in its response.

```
 // This agent has determined that it needs the middle name of a student.
 // It knows the provider of the student is the “SASIxp” agent because it
 // has recently received a SIF_Event from that agent. This code sends a
 // directed query to the “SASIxp” agent for a specific StudentPersonal
 // object identified by RefId. An array of fields is passed to the
 // Query.setFieldRestrictions method to instruct the recipient to only
 // return the <Name> element of the student.

 Query q = new Query( SIFDTD.STUDENTPERSONAL );

 // Add a condition to query for a specific RefId
 q.addCondition( SIFDTD.STUDENTPERSONAL_REFID, Condition.EQ,
“4A37969803F0D00322AF0EB969038483”);

 // Add optional field restrictions
 q.setFieldRestrictions( new ElementDef[] {
   SIFDTD.STUDENTPERSONAL_NAME
 };

 // Send the query to the “SASIxp” agent in this zone
 Zone myZone = getZoneFactory().getZone( “MYZONE” );
 myZone.query( q, “SASIxp”, 0 );
```

Handling Query Results (SIF_Response Messages)

SIF’s request and response model is asynchronous, which means the results of a query will be received by your agent at any time as `SIF_Response` messages are retrieved from its queue on the server. If the agent responding to a query is running, connected to the zone, and able to process `SIF_Request` messages quickly, responses may be received soon after the query is sent. On the other hand, if the responding agent is down or is not able to process the query at the time it is received, your agent may not receive `SIF_Response` messages until long after the query was issued.

The QueryResults Interface

The *QueryResults* interface is implemented by classes that can handle `SIF_Response` messages received in response to a query issued by the agent. Unlike the *Publisher* and *Subscriber* interfaces, which have one method apiece, *QueryResults* has two methods:

```
public void onQueryPending( MessageInfo info, Zone zone )
  throws ADKException;

public void onQueryResults(
  DataObjectInputStream data,
  SIF_Error error,
  Zone zone,
  MessageInfo info )
    throws ADKException;
```

The `onQueryPending` method is called by the ADK when it *successfully* sends a `SIF_Request` message to a zone. This method allows your agent to keep track of the message IDs of each request that has been issued. If tracking the requests sent by your agent is not necessary to its operation, this method can be unimplemented.

The `onQueryResults` method is called by the ADK when a `SIF_Response` message is received.

Implementing onQueryResults

The DataObjectInputStream Parameter

The first parameter is onQueryResults is an input stream from which you can enumerate the `SIFDataObjects` received by the ADK in a SIF_Response message. To do so, repeatedly call the `DataObjectInputStream.readDataObject` method until no more objects are available. This is done in a *while* loop as illustrated below.

```
public void onQueryResults(
  DataObjectInputStream data,
  SIF_Error error,
  Zone zone,
  MessageInfo info )
    throws ADKException
{
  // Check for a SIF_Error condition
  if( error != null )
  {
    System.out.println( "The query failed: " + error );

    return;
  }

  // Continue processing SIFDataObjects from the input stream as long
  // as the available method returns true
  while( data.available() )
  {
    // Get the next SIF Data Object
    SIFDataObject object = data.readDataObject();

    // Do something with the data...
  }
}
```

If your *QueryResults* message handler is registered to a specific SIF Data Object type, your code can assume that the type of object contained in SIF Response messages passed to that handler are of a certain type. For example, suppose you've registered a *QueryResults* handler with a zone specifically for the `SIFDTD.STUDENTPERSONAL` object type:

```
// Register a QueryResults message handler with the zone to handle
// responses to queries for students only
myZone.setQueryResults( SIFDTD.STUDENTPERSONAL, myStudentDatabase );
```

In this case, you can assume that all `SIFDataObject` instances returned by the input stream will be `openadk.library.student.StudentPersonal` instances and can safely cast objects to that type. However, suppose you have registered a single *QueryResults* implementation with a zone to handle all `SIF_Response` messages regardless of payload:

```
// Register a single QueryResults message handler with the zone to handle
// responses to all queries regardless of object type
myZone.setQueryResults( this );
```

In this case, it is usually necessary to determine the type of object that will be returned by the input stream's `readDataObject` method. This can be done by calling `DataObjectInputStream.getObjectType`:

```
public void onQueryResults(
  DataObjectInputStream data,
  SIF_Error error,
  Zone zone,
  MessageInfo info )
    throws ADKException
{
  // Check for a SIF_Error condition
  if( error != null )
  {
    System.out.println( "The query failed: " + error );

    return;
  }

  // What kind of SIFDataObject will be returned by the input stream?
  ElementDef type = data.getObjectType();

  while( data.available() )
  {
    // Get the next SIF Data Object
    SIFDataObject object = data.readDataObject();

    // Handle it appropriately based on object type
    if( type == SIFDTD.STUDENTPERSONAL )
    {
      StudentPersonal sp = (StudentPersonal)object;

      // Do something with the student...
    }
    else
    if( type == SIFDTD.STAFFPERSONAL )
    {
      StaffPersonal sp = (StaffPersonal)object;

      // Do something with the teacher...
    }
  }
}
```

Notice the `DataObjectInputStream.getObjectType` method returns a `SIFDTD` constant to uniquely identify the type of object in a version-independent way. This approach is used consistently throughout the ADK because the string names of elements and attributes in SIF may change from version to version. By using `SIFDTD` constants your code is portable from one version of SIF to the next.

ⓘ If you're not familiar with the SIFDTD class, please refer to Creating & Manipulating SIFDataObjects on page 20.

The ADK supplies two ways to create and manipulate `SIFDataObject` instances. The first is to programmatically construct objects and call the methods of the SDO classes to get and set element and attribute values. For example,

```
// Constructing a StudentPersonal object
StudentPersonal sp = new StudentPersonal();
sp.setRefId( refid );
sp.setName( NameType.LEGAL, “Johnson”, “Clifford” );

// Examining a StudentPersonal object
Name n = sp.getName();
System.out.println( “Name: “ + n.getLastName() + “, “ + n.getFirstName() );
OtherId[] ids = sp.getOtherIds();
System.out.println( “This student has “ + ids.length + “ IDs” );
```

You can also use the `setElementOrAttribute` method:

```
// Dynamically constructing a StudentPersonal object
SIFDataObject sp = ADK.DTD().createSIFDataObject( SIFDTD.STUDENTPERSONAL );
sp.setElementOrAttribute( “@RefId”, refid );
sp.setElementOrAttribute( “Name[@Type=’01’]/LastName”, “Johnson” );
sp.setElementOrAttribute( “Name[@Type=’01’]/FirstName”, “Clifford” );

// Dynamically examining a StudentPersonal object
String refId = sp.getElementOrAttribute( “@RefId” );
String studentId = sp.getElementOrAttribute( “OtherId[@Type=’06’” );
```

An alternative and more data-driven approach to working with `SIFDataObjects` is to use the `Mappings` class from the `openadk.library.tools.Mappings` package to convert `SIFDataObject` instances to and from a `HashMap` by applying a set of XPath-like mapping rules. These rules are usually stored in an external configuration file where they can be modified by system integrators in the field. For example, the following rules for StudentPersonal objects define how to translate the elements and attributes of that object to a flat list of field values:

```
<object object=”StudentPersonal”>
       <field name=”STUDENT_ID”>OtherId[@Type=’01’]</field>
       <field name=”LAST_NAME”>Name[@Type=’01’]/LastName</field>
       <field name=”FIRST_NAME”>Name[@Type=’01’]/FirstName</field>
</object>
```

You can then call upon the `Mappings` class to convert a StudentPersonal instance into a `HashMap` of field/value pairs, or to convert a `HashMap` into a StudentPersonal object.

In practice, most agents use a combination of these two approaches when working with SIF Data Objects. Refer to the chapter on SIF Data Objects for more information

The SIF_Error Parameter

The second parameter to `onQueryResults` is a `SIF_Error` instance from the `openadk.library.infra` package. This parameter is null if the query was processed successfully by the responding agent. However, if the responder encountered an error while processing the query, it will return a `SIF_Error` element in its `SIF_Response` message. In this case the *error* parameter will be non-null.

It is good practice to always check the error parameter before enumerating SIF-DataObjects from the input stream.

The Zone Parameter

The third parameter to `onQueryResults` is a *Zone* object identifying the zone from which the `SIF_Response` message was received. You can use this information to select the appropriate database to update with the objects received by the query.

The MessageInfo Parameter

The final parameter to `onQueryResults` is a `MessageInfo` object, which provides miscellaneous information about the message such as its header fields, message ID, and the *SourceId* of the sender. These attributes can be retrieved by casting the object to a `SIFMessageInfo`. This parameter can also be used to retrieve attributes specific to `SIF_Response` messages, including:

- The message identifier of the original `SIF_Request` message. Call the `SIFMessageInfo.getSIFRequestMsgId` method to obtain this value.

- The `SIF_Response` packet number. Call the `SIFMessageInfo.getPacketNumber` method to obtain this value.

- Whether or not this is the last `SIF_Response` message in a series. Call the `SIFMessageInfo.getMorePackets` method to obtain this value.

- Retrieve any state information that you might persisted with the request. To do so, call `SIFMessageInfo.getRequestInfo()` to retrieve a `RequestInfo` instance. To retrieve the state information, simply call `RequestInfo.getUserData()`. `RequestInfo` also tracks the date and time of the original request, which you could retrieve for statistical purposes.

When used in conjunction with the `QueryResults.onQueryPending` method, the above information can be useful in tracking the completion of queries issued by the agent. Tracking queries is optional but recommended for agents that have a user interface and need to keep users informed of the progress of queries that have been issued to date.

Refer to the Javadoc for more information on the `SIFMessageInfo` class.

Tracking the Progress of Queries

To track the progress of queries, implement `QueryResults.onQueryPending` to record the message identifier of each `SIF_Request` message sent to a zone. This is easily done in-memory by using a temporary table such as a `HashMap`. However, because the SIF Request & Response model is asynchronous, you should consider using a persistent file or database table versus an in-memory cache if this information is important to the operation of your agent.

When incoming `SIF_Responses` are received by the `QueryResults.onQueryResults` method, call `SIFMessageInfo.getSIFRequestMsgId` to determine which query the response is associated with. Next, call the `SIFMessageInfo.getMorePackets` function to determine if the response is the last in a series. If so, the agent knows it has successfully received all data from that query.

Querying in Response to SIF Events

If the processing of a `SIF_Event` will require that your agent query for additional data in order to complete the operation, construct a `TrackQueryResults` object to execute a query and block the current thread until the results are received. When using `TrackQueryResults` in this way, you must pass the `Event` object to its constructor so the ADK can send an “Intermediate” acknowledgement to the ZIS, thereby invoking Selective Message Blocking. When the request has been satisfied, the `TrackQueryRes-`

ults object will automatically send a “Final” acknowledgement to remove the block from the SIF_Event message.

11. Using SIFEncryption

The SIF 1.5 Authentication object allows usernames and passwords to be shared among applications. To further secure the password, SIF requires that the password be either obscured using BASE64 encoding or encrypted or hashed using a standard set of cryptographic algorithms. The ADK’s SIFEncryption class manages the complexity of dealing with encrypted and obscured passwords and dealing with shared keys between SIF-enabled applications.

Supported Algorithms

The SIF Specification enumerates the algorithms that can be used to obscure, hash, or encrypt passwords. The OpenADK supports the following algorithms from that list:

- **Base64.** The passwords are encoded using the BASE64 encoding algorithm. Note that in this mode, passwords are not encrypted and can be easily retrieved using a BASE64 decoder.

- **SHA1 and MD5.** These algorithms create a “hash” of the password, which can be used to authenticate a user by hashing the password that they use at login time and comparing it to this value.

- **DES, TripleDES, and RC2.** These algorithms are known as symmetric encryption algorithms and all for the password to be securely encrypted and decrypted using a key that is shared between the applications. The actual encryption key is not shared using SIF. The key must be pre-configured between the applications by an administrator. The password cannot be decrypted without knowledge of the secret key.

Usage by Subscriber Agents

To use the SIFEncryption class to read passwords from an Authentication object provider, you should typically call the SIFEncryption.getInstance() overload that accepts a AuthencationInfoPassword instance and a Zone. This method will create and initialize the appropriate SIFEncryption instance and return it. If the provider is using one of the symmetric encryption methods, such as DES, TripleDES, or RC2, as shown in the example object below, the private key needs to configured in the agent or zone properties. This can be done either by modifying the agent’s configuration file as shown below (if you are using the openadk.library.tools.AgentConfig class) or by calling agent.getProperties().setEncryptionKey(keyName, key). An example of using the agent’s configuration file is shown below.

```
<!-- The 128-bit key used for sending passwords -->
<property name="adk.encryption.keys.SECRET_64-BIT_KEY" value="dW7SKzwdn0Q=" />
```

If the <password> element is using an encryption method that does not require a key, such as base64, SHA1, or MD5, it is not using an encryption key. The KeyName attribute is ignored, and the SIFEncryption class will not look for a key in the agent properties by that name. To read the password, simple call `readPassword( AuthencactionInfoPassword )` method. If the password was stored as clear-text (base64) or encrypted, the plain-text password will be returned. If the password was stored as a hash value, the hash value is returned, as is, from the element.

Here is an example Authentication object that might be received from a provider. Note that the `KeyName` attribute references a key that must be present in the agent properties for the `SIFEncryption` class to automatically find it.

```
<Authentication RefId="558B19E4A85843A5B7ACB193BF8A190D"
    SifRefId="82B23671D6204FFE9E9C154E87017F83"
      SifRefIdType="StudentPersonal">
      <AuthenticationInfo>
            <System Type="Application">Sample SIF Application</System>
            <Username>example_user</Username>
            <DistinguishedName>cn=Example User, cn=Users, dc=sifinfo,
dc=org</DistinguishedName>
            <Password Algorithm="DES" KeyName="SECRET_64-BIT_KEY">
                 6XSjrzAgkrd41Nzb61w5vwuqzKsQbybL</Password>
      </AuthenticationInfo>
</Authentication>
```

Here's an example of a subscriber agent reading a password field:

```
  private void handleAuthenticationReceived(Authentication auth, Zone zone) {
        AuthenticationInfo[] infos = auth.getAuthenticationInfos();
        for (int a = 0; a < infos.length; a++) {
            try {
                SIFEncryption decryptor = SIFEncryption.getInstance(infos[a]
                        .getPassword(), zone);
                System.out
                        .println("Received AuthenticationInfo/Password " +
                                + " using algorithm "
                                + decryptor.getAlgorithm().value
                                + " for user "
                                + infos[a].getUsername()
                                + ", password="
                                + decryptor
                                      .readPassword(infos[a].getPassword()));
            } catch (NoSuchAlgorithmException nsae) {
                System.out
                        .println("Received Password using algorithm "
                                + infos[a].getPassword().getAlgorithm()
                                + " for user "
                                + infos[a].getUsername()
                                + "but the cipher algorithm is not available");
            } catch (Exception ex) {
                System.out.println("Error processing object: "
                      + ex.toString());
            }
            System.out.println();
        }
    }
```

Usage by Provider Agents

A provider of Authentication objects can choose to support one or more than one encryption method when sending passwords using the ADK. The simplest manner of providing password information is to start by specifying the default algorithm by calling `AgentProperties.setDefaultEncryptionAlgorithm(String algorithm)` or setting the property `adk.encryption.algorithm` in the agent's configuration file. Setting the value in the configuration file is more flexible and will allow the agent to adapt to subscribers that expect passwords in a specific format.

To instantiate an instance of SIFEncryption for writing passwords, simply call the `getInstance( Zone )` overload. This method searches for two properties in the agent properties.

- `adk.encryption.algorithm` returns the default algorithm the agent uses for encryption
- `adk.encryption.key` returns the name of the key to use for encryption, which, if required, will be read from the `adk.encryption.keys.[keyName]` property.

The `adk.encryption.keys.[keyName]` property is not used or required for base64, SHA1, and MD5. The `adk.encryption.key` property is required, even for encryption methods that do not use a key. It is this value that will be written to the `KeyName` attribute of the AuthenticationInfoPassword element.

Here is an example of an agent responding to a request for Authentication objects and returning a single object. Depending on what algorithm was specified in the agent properties, the result could be published as Base64, SHA1, MD5, RC2, DES, or TripleDES.

```
public void onRequest(DataObjectOutputStream out,
    Query query,
    Zone zone,
    MessageInfo info)
    throws ADKException
{
   try
   {
    AuthenticationInfo inf  = new AuthenticationInfo(
      new AuthenticationInfoSystem(
        SystemType.APPLICATION,
        "Sample SIF Application"));
    inf.setDistinguishedName("cn=Example User,cn=Users,dc=sifinfo,dc=org");
    inf.setUsername( "example_user" );
    inf.setPassword( new AuthenticationInfoPassword() );

    try
    {
      SIFEncryption encryptor = SIFEncryption.getInstance(zone);
      encryptor.writePassword(inf.getPassword(), "secret");
    }
    catch( Exception ex )
    {
      System.out.println( ex.toString());
    }

    Authentication auth = new Authentication(
      ADK.makeGUID(),
      ADK.makeGUID(),
      AuthSifRefIdType.STAFFPERSONAL,
      inf);
    out.write(auth);
  }
  catch( Exception ex )
  {
       System.out.println(ex.toString());
  }
}
```

Limitations

The encryption algorithms available in the SIFEncryption class are dependent upon the ciphers available in the version of Java that is being used. Please refer to the table below to determine which encryption algorithms are available, by default, in each version.

Java Version	Algorithms Supported
Java 2 versions 1.2.x and 1.3.x [requires Java Cryptography Extension (JCE) 1.2.2]	Base64, SHA1, MD5, DES, TripleDES
Java 2 version 1.4.2.x	Base64, SHA1, MD5, DES, TripleDES
Java 5	Base64, SHA1, MD5, DES, TripleDES, and RC2

NOTE: In this release, the OpenADK does not support the RSA or DSA encryption algorithms defined by SIF.

12. Appendix A Revisions

What's New in 1.5.2.0

Enhancements to Request/Response

Setting Query State

The Query class now allows you to associate a small piece of state information with the query by calling `Query.setUserData`. The only requirement of your state object is that it needs to implement the `Serializable` interface. This state information will be persisted by the ADK and available later on when the response messages are received.

Retrieving Query State

The state item that was stored when the `Query` was created can be retrieved in the `SIFMessageInfo` class that is returned in the response. To do so, call `SIFMessageInfo.getRequestInfo()` to retrieve a `RequestInfo` instance. To retrieve the state information, simply call `RequestInfo.getUserData()`. The `RequestInfo` class also tracks the date and time of the original request, which you can retrieve for statistical purposes.

What's New in 1.5.1.5

Data Model Classes

SIFElement Enhancements

The SIFElement class has some new methods that have been added to increase its functionality:

- `setChildren( ElementDef, Element[] )` adds the specified children as an array to the SIFElement object. All existing children that are defined as the same type as the ElementDef parameter are removed and replaced with this list. Calling this method with the Element[] parameter set to null removes all children.
- `addChild()` methods were changed so that if the same child is added twice, it will not throw an Exception.
- `getId()` and `setId( string )` methods were added to allow the application to get and set a unique identifier for each SIFElement. These values are meant to support external object persistence frameworks and are not used internally by the ADK.
- If `setField( ElementDef id, String value )` is called with a null value, the field is removed from the list.

Changes to SIF Data Objects

- A new setter method was added for each repeatable element in all SIF data objects that accepts an array of objects. For example, the method `StudentPersonal.-setEmails( Email[] emails )` was added. This method completely replaces the existing repeatable elements with the new list. If called with a null value, the repeatable elements are all removed.
- A new method, `setSIF_ExtendedElementsContainer( SIF_ExtendedElements )` was added to the SIFDataObject class.

Changes to Enum

The Enum class was changed to override the `equals()` and `hashcode()` methods so that Enum instances can be compared directly.

The SIFEncryption Class

The SIF Authentication object allows usernames and passwords to be shared among applications. To further secure the password, SIF requires that the password be either obscured using BASE64 encoding or encrypted or hashed using a standard set of cryptographic algorithms. The new SIFEncryption class in the OpenADK manages the complexity of dealing with encrypted and obscured passwords and dealing with shared keys between SIF-enabled applications.

Supported Algorithms

The SIF specification specifies a list of algorithms that can be used to obscure, hash, or encrypt passwords. The ADK supports the following algorithms from that list.

- **Base64.** The passwords are encoded using the BASE64 encoding algorithm. Note that in this mode, passwords are not encrypted and can be easily retrieved using a BASE64 decoder.
- **SHA1 and MD5**. These algorithms create a “hash” of the password, which can be used to authenticate a user by hashing the password that they use at login time and comparing it to this value.
- **DES, TripleDES, and RC2**. These algorithms are known as symmetric encryption algorithms and all for the password to be securely encrypted and decrypted using a key that is shared between the applications. The actual encryption key is not shared using SIF. The key must be pre-configured between the applications by an

administrator. The password cannot be decrypted without knowledge of the secret key.

Usage by Subscriber Agents

To use the SIFEncryption class to read passwords from an Authentication provider, you should typically call the `SIFEncryption.getInstance( AuthencationInfoPassword, Zone)` method. This method will create and initialize the appropriate instance of the SIFEncryption class and return it. If the provider is using one of the symmetric encryption methods, such as DES, TripleDES, or RC2, as shown above, the private key needs to configured in the agent or zone properties. This can be done either by modifying the agent's configuration file as shown below (if you are using the `openadk.library.tools.AgentConfig` class) or by calling `agent.getProperties().setEncryptionKey( keyName, key )`. An example of using the agent's configuration file is shown below.

```
<!-- The 128-bit key used for sending passwords -->
<property name="adk.encryption.keys.SECRET_64-BIT_KEY" value="dW7SKzwdn0Q=" />
```

If the `<password>` element is using an encryption method that does not require a key, such as base64, SHA1, or MD5, it is not using an encryption key. The KeyName attribute is ignored, and the SIFEncryption class does not look for a key in the agent properties by that name. To read the password, simple call `readPassword(AuthencactionInfoPassword)`. If the password was stored as clear-text (BASE64) or encrypted, the plain-text password will be returned. If the password was stored as a hash value, the hash value is returned, as is, from the element.

Here is an example Authentication object that might be received from a provider. Note that the KeyName attribute references a key which must be present in the agent properties for the SIFEncryption class to automatically find it.

```
<Authentication RefId="558B19E4A85843A5B7ACB193BF8A190D"
                SifRefId="82B23671D6204FFE9E9C154E87017F83"
        SifRefIdType="StudentPersonal">
        <AuthenticationInfo>
                <System Type="Application">Sample SIF Application</System>
                <Username>example_user</Username>
                <DistinguishedName>cn=Example User, cn=Users, dc=sifinfo, dc=org</DistinguishedName>
                <Password Algorithm="DES" KeyName="SECRET_64-BIT_KEY">6XSjrzAgkrd41Nzb61w5vwuqzK-
sQbybL</Password>
   </AuthenticationInfo>
</Authentication>
```

Here is an example of a subscriber agent reading a password field.

```
private void handleAuthenticationReceived(
        Authentication auth, Zone zone )
{
        AuthenticationInfo[] infos = auth.getAuthenticationInfos();
  for (int a = 0; a < infos.length; a++) {
                try {
        SIFEncryption decryptor =
                                SIFEncryption.getInstance(infos[a].getPassword(), zone);
    System.out.println(
                                "Received AuthenticationInfo/Password using algorithm "
                + decryptor.getAlgorithm().value
        + " for user "
        + infos[a].getUsername()
        + ", password="
        + decryptor.readPassword(infos[a].getPassword()));
                } catch (NoSuchAlgorithmException nsae) {
        System.out.println(
                                "Received AuthenticationInfo/Password using algorithm "
                                        + infos[a].getPassword().getAlgorithm()
        + " for user "
        + infos[a].getUsername()
        + "but the cipher algorithm is not available on the system");
```

```
                        } catch (Exception ex) {
            System.out.println("Error processing object: " + ex.toString());
                        }
                        System.out.println();
            }
}
```

Usage by Provider Agents

A provider of Authentication can choose to support one or more than one encryption method when sending passwords using the ADK. The simplest manner of providing password information is to start by specifying the default algorithm by calling `AgentProperties.setDefaultEncryptionAlgorithm(String algorithm)` or setting the property `adk.encryption.algorithm` in the agent's configuration file. Setting the value in the configuration file is more flexible and will allow the agent to adapt to subscribers that expect passwords in a specific format.

To instantiate an instance of SIFEncryption for writing passwords, simply call the `getInstance(Zone)` overload. This method searches for two properties in the agent properties.

- `adk.encryption.algorithm` returns the default algorithm the agent uses for encryption
- `adk.encryption.key` returns the name of the key to use for encryption, which, if required, will be read from the `adk.encryption.keys.[keyName]` property.

The `adk.encryption.keys.[keyName]` property is not used or required for base64, SHA1, and MD5. The `adk.encryption.key` property is required, even for encryption methods that do not use a key. It is this value that will be written to the `@KeyName` attribute of the AuthenticationInfoPassword object.

Here is an example of an agent responding to a request for Authentication and returning a single object. Depending on what algorithm was specified in the agent properties, the result could be published as Base64, SHA1, MD5, RC2, DES, or TripleDES.

```
public void onRequest(DataObjectOutputStream out,
                        Query query,
                        Zone zone,
                        MessageInfo info)
                        throws ADKException {
  try
  {
                        AuthenticationInfo inf  = new AuthenticationInfo(
                                    new AuthenticationInfoSystem(
                                                SystemType.APPLICATION,
                                                "Sample SIF Application"));
                        inf.setDistinguishedName( "cn=Example User, cn=Users, dc=sifinfo, dc=org" );
                        inf.setUsername( "example_user" );
                        inf.setPassword( new AuthenticationInfoPassword() );
                        try
                        {
                                    SIFEncryption encryptor = SIFEncryption.getInstance(zone);
                                    encryptor.writePassword(inf.getPassword(), "secret");
                        }
                        catch( Exception ex ) {
                                    System.out.println(ex.toString());
                        }

                        Authentication auth = new Authentication(
                                    ADK.makeGUID(),
                                    ADK.makeGUID(),
                                    AuthSifRefIdType.STAFFPERSONAL,
                                    inf);

                        out.write(auth);
  }
  catch( Exception ex ) {
                          System.out.println(ex.toString());
              }
}
```

Limitations

The encryption algorithms available in the SIFEncryption class are dependent upon the ciphers available in the version of Java that is being used. Please refer to the table below to determine which encryption algorithms are available, by default, in each version.

Java Version	Algorithms Supported
Java 2 versions 1.2.x and 1.3.x [requires Java Cryptography Extension (JCE) 1.2.2]	Base64, SHA1, MD5, DES, TripleDES
Java 2 version 1.4.2.x	Base64, SHA1, MD5, DES, TripleDES
Java 5	Base64, SHA1, MD5, DES, TripleDES, and RC2

NOTE: In this release, the OpenADK does not support the RSA or DSA encryption algorithms defined by SIF.

What's New in 1.5.1.0

SIF 1.5 Infrastructure Support

The ADK supports three new features of the SIF 1.5 infrastructure:

- SIF_ExtendedElement elements
- SIF_LogEntry objects
- SIF_SystemControl/SIF_GetZoneStatus messages (experimental)

SIF_ExtendedElement

Beginning with SIF 1.5, users can add custom extensions to any SIF Data Object by appending a SIF_ExtendedElements element to the end of the object. This element serves as a container for one or more SIF_ExtendedElement children, which can be used to pass mutually-defined field values among applications, experiment with extensions to the SIF Data Object model, or enhance objects for custom integration projects.

To set the value of an extended element, call the new `setExtendedElement` method on any SIFDataObject instance:

```
StudentPersonal sp = new StudentPersonal();
sp.setRefId( "…" );
sp.setExtendedElement( "InternetUsePolicy", "accept" );
sp.setExtendedElement( "Thumbprint", "19zl+aFFNa0192…" );
sp.setName( new Name( NameType.BIRTH, "Smith","Joe" ) );
sp.setOtherId( OtherIdType.SYSTEM, "1600001" );
```

The above calls would result in this StudentPersonal object:

```
<StudentPersonal RefId="…">
   <OtherId Type="06">1600001</OtherId>
   <Name Type="01">
      <LastName>Smith</LastName>
      <FirstName>Joe</FirstName>
   </Name>
   <SIF_ExtendedElements>
      <SIF_ExtendedElement name="InternetUsePolicy">
         accept
      </SIF_ExtendedElement>
      <SIF_ExtendedElement name="Thumbprint">
         19zl+aFFNa0192…
      </SIF_ExtendedElement>
   </SIF_ExtendedElements>
</StudentPersonal>
```

To extract all SIF_ExtendedElements from a SIFDataObject, call this new method:

```
public SIF_ExtendedElement[] getExtendedElements();
```

To lookup a specific SIF_ExtendedElement by name, call this method:

```
public SIF_ExtendedElement getExtendedElement( String name );
```

For example, the following *Subscriber.onEvent* implementation for StudentPersonal objects first looks for a "InternetUsePolicy" extended element and saves its value to a local variable. It then builds a Hashtable of all extended elements found in the StudentPersonal object.

```
public void onEvent( Event ev, Zone zone, MessageInfo info )
   throws ADKException
{
   DataObjectInputStream data = ev.getData();
   while( data.isAvailable() )
   {
      // Get next StudentPersonal from the input stream
      StudentPersonal sp = (StudentPersonal)data.readDataObject();

      // Look for InternetUsePolicy extended element
      String usePcy = sp.getExtendedElement( "InternetUsePolicy" );
      if( usePcy != null && usePcy.equals("accept") )
      {
         // Do something with this value...
      }

      // Put all SIF_ExtendedElements in a hashtable
      Hashtable extras = new Hashtable();
      SIF_ExtendedElement[] ext = sdo.getSIFExtendedElements();
      if( ext != null ) {
         for( int i = 0; i < ext.length; i++ ) {
            String name = ext[i].getName();
            String value = ext[i].getTextValue();
            extras.put( name, value );
         }
      }
   }
}
```

SIF_SystemControl/SIF_GetZoneStatus

By sending this message to a ZIS, an agent can obtain zone status synchronously. This alleviates some of the chicken-egg problems vendors have experienced in obtaining

SIF_ZoneStatus objects with the asynchronous request and response model (for example, an agent needs SIF_ZoneStatus to make logic decisions but pending messages in the agent's queue prevent it from receiving that object in time.)

To use this feature, set the "useZoneStatusSystemControl" agent property to a value of true in your agent's **initialize** method as shown below. This property is disabled by default.

```
AgentProperties props = getProperties();
props.setUseZoneStatusSystemControl( true );
```

The above calls instruct the `Zone.getZoneStatus` method to issue a SIF_SystemControl message to obtain a zone status synchronously instead of sending a SIF_Request and blocking the current thread until a SIF_Response is received.

NOTE: This feature is considered experimental by SIF and may not be implemented in the specification. If adopted, we will reverse the default value of this property in a future ADK so that it is enabled by default in the version it appears in the specification.

The ServerLog Class

The SIF_LogEntry Object

SIF 1.5 introduces the new SIF_LogEntry data object to provide agents with a means of reporting diagnostic and error/warning messages back to a zone. These log entries are sent independent of the SIF_Ack acknowledgements that accompany each SIF_Message. They may be comprised of simple textual messages or can be more complex in structure, referencing SIF_Messages and the data objects contained in them.

SIF_LogEntry is particularly useful when an agent processes a message at some point in time after it has received the message from the server. In earlier versions of SIF, there was no mechanism to report log information to a zone other than by returning a SIF_Ack with an error code and description, which could only be accomplished at the time the message was received by the agent. SIF_LogEntry addresses that limitation. Another benefit of SIF_LogEntry is that it can reference SIF_Messages and SIF Data Objects in its payload. Subscribers can use that information to reconstruct the path of a message or data object from its publisher to the agents that received it, promoting the development of log analysis features in zone integration servers.

SIF_LogEntry is designed to follow the normal publish and subscribe model of SIF. When the ADK reports a SIF_LogEntry object to a zone, the zone integration server will broadcast it to all subscribers. Similarly, agents can query a zone integration server for log information by issuing a SIF_Request for SIF_LogEntry objects. By design, zone integration servers usually provide built-in support for this object type so that log information can be recorded and published without any third-party agents. In the ZIS, for example, all SIF_LogEntry objects reported to a zone are written to the agent's log file on the server.

The ServerLog Class

The ADK 1.5 introduces the concept of a "server log" to which SIF_LogEntry messages can be written. Every zone has a server log that represents a virtual log on the zone integration server. Call the new `Zone.getServerLog()` function to obtain the ServerLog instance for a zone, then call one of its `log` methods to send a SIF_LogEntry object to the server. The version of the method you call depends on how much information you want to send to the zone integration server, and whether or not that information references SIF Messages and SIF Data Objects received by the agent.

When reporting log information that references a SIF Message and set of data objects previously received by the agent, you can pass as parameters the *SIFMessageInfo* instance from your message handler, and optionally an array of data objects to reference in the log. The `log` method will package up the required information into a SIF_LogEntry object, and then report it to the zone via an Add event.

Since SIF_LogEntry is available only in SIF 1.5 and later, the ADK will not attempt to send these objects if the agent is running in an earlier version. Instead, it will echo the information to the local agent log. This enables you to code SIF_LogEntry into your agent without having to worry about the version of SIF the agent is running against.

Logging Message-Independent Entries

To report log information independent of a SIF Message or SIF Data Objects – for example, a notice to the zone that the agent has completed its synchronization process; has run out of memory and will shut down; etc. – call the following forms of `log`. These methods do not require that you pass any information about SIF Messages or SIF Data Objects.

```
public void log(String message );

public void log(
    LogLevel level,
    String desc,
    String extDesc,
    String appCode );

public void log(
    LogLevel level,
    String desc,
    String extDesc,
    String appCode,
    int category,
    int code );
```

Logging Message-Dependent Entries

To report log information that references a SIF Message previously received by the agent – for example, to report that a business rule has failed because the agent did not receive enough information in a SIF_Event message – call the following forms of `log`. These methods are identical to the above except they allow a SIFMessageInfo and array of SIFDataObjects to be passed as parameters. You may pass a null value to the `SIFMessageInfo` and `SIFDataObject[]` parameters. If the `SIFDataObject[]` parameter is non-null, the `SIFMessageInfo` parameter must be supplied.

```
public void log(
    LogLevel level,
    String message,
    SIFMessageInfo info,
    SIFDataObject[] objects );

public void log(
    LogLevel level,
    String desc,
    String extDesc,
    String appCode,
    SIFMessageInfo info,
    SIFDataObject[] objects );

public void log(
    LogLevel level,
    String desc,
    String extDesc,
    String appCode,
    int category,
    int code,
    SIFMessageInfo info,
    SIFDataObject[] objects );
```

New Features in the Mappings Classes

Automatic ValueSet Translation

The ValueSet class from the `openadk.library.tools.mappings` package offers a way to build translation tables for converting codes and other constants used by an application to codes used by SIF. For example, many agents must convert between the standard Ethnicity, Language, and Grade Level codes defined in the SIF Specification to equivalent codes used by the application. ValueSets let you build these translation tables in your configuration file when using the Mappings classes.

In this release of the ADK, the Mappings classes have been enhanced to automatically translate values if a *ValueSet* attribute is assigned to a FieldMapping. For example, consider the following mapping rules in a configuration file:

```
<mappings id="Default">
       <valueset id="LanguageCodes">
              <value name="EN" title="English">ENG</value>
              <value name="SP" title="Spanish">SPA</value>
       </valueset>
       <object object="StudentPersonal">
              <field name="LANG" valueset="LanguageCodes">
                     Demographics/Language
              </field>
       </object>
</mappings>
```

The new *valueset* attribute of the <field> rule instructs the `Mappings.map` function to automatically lookup the ValueSet with an ID of "LanguageCodes", and if found translate the value of the Demographics/Language element using that table. The `Mappings.map` function calls `ValueSet.translate` or `ValueSet.translateReverse` depending on which whether the mapping is for an inbound or outbound message.

Deferred Responses

By design, the ADK sends SIF_Response messages to the zone integration server after it processes the SIF_Request and control has returned from the `Publisher.onRequest` message handler. From the ADK perspective, the two messages are considered part of the same logical operation. The new Deferred Response capability in ADK 1.5 lets agent developers decouple SIF_Request processing from the sending of SIF_Response messages.

Overview of Request & Response Handling in the ADK

When the ADK receives a SIF_Request message from a zone, it delegates the processing of that message to a *Publisher* message handler selected from the message dispatch chain. The `onRequest` method is called and supplied with a DataObjectOutputStream instance, to which one or more SIFDataObject instances and/or SIF_Error elements may be streamed.

As objects are written to this stream, the ADK renders them and packages them into individual SIF_Response packets. This is done according to the SIF Version, SIF_MaxBufferSize, and SIF_Element field restrictions specified in the SIF_Request message. These packets are written to the local file system in the agent's work directory pending delivery to the server. When `onRequest` returns successfully, a background thread is started to deliver the packets to the zone integration server in one or more SIF_Response messages. The entire process occurs as one logical operation, with a SIF_Request always met with one or more SIF_Responses.

Deferring Responses

While the ADK's normal request/response behavior is very easy to use and appropriate for most agents, there are times when it is necessary to decouple SIF_Request and SIF_Response processing. For example, some agents may wish to record SIF_Requests in a work queue and then processes them at a later time when data is available. In this case, the ADK must not return SIF_Response messages to the requestor when control returns from the `Publisher.onRequest` method. Another example might be an agent that implements the SIF 1.5 StudentLocator message choreography, where SIF_Responses are usually sent at some point in the future after the initial SIF_Request for StudentLocator is received.

Deferred Responses is a new feature of the ADK whereby agents can "defer" SIF_Response messaging by calling the `deferResponse` method of the DataObjectOutputStream. Calling this method instructs the ADK to ignore any SIFDataObjects that are written to the stream, and to not send any SIF_Responses to the zone integration server when control is returned from the `onRequest` method. By calling `deferResponse`, you're in effect telling the framework that you will handle sending SIF_Responses at a later time.

TODO

When the agent is ready to send SIF_Responses, it uses the new SIFResponseSender class in the `openadk.library` package to stream SIFDataObjects and/or SIF_Error elements to the zone.

The SIFResponseSender Class

The SIFResponseSender class follows the same general design and API as the Publisher.onRequest method for sending SIF_Response packets to a zone integration server. When deferring responses as described above, an agent can use this class when it is ready to send SIF_Response packets. Follow these steps:

1. Construct an instance of SIFResponseSender and call its `open` method

2. Stream one or more SIFDataObject instances by calling the `write` method. This is similar to what an agent would normally do in the `Publisher.onRequest` meth-

od when data is streamed to the supplied DataObjectOutputStream. As with traditional processing, packets are rendered and cached on the local file system pending delivery to the server.

3. To include a SIF_Error element in the response stream, call the version of `write` that accepts a SIF_Error instance.

4. When you're finished streaming data objects and/or a SIF_Error element to the SIFResponseSender, call its `close` method. The ADK will begin delivering the SIF_Response packets to the zone integration server. Note as with traditional processing, this is handled in a background thread. If the agent shuts down before all responses have been delivered it will automatically pick up where it left off the next time the agent is started.

Miscellaneous

SIF_URL Hostname & Port

When registering with a zone in Push mode, the ADK binds to a local IP address and port to listen for incoming messages from the zone integration server. It uses the Host and Port properties from the TransportProperties object when binding the socket. You set these values by obtaining the HttpProperties object from the agent and then calling the `setHost` and `setPort` methods as needed. The SIF_Protocol/SIF_URL element included with each zone's SIF_Register message is automatically prepared by the ADK in the following format, using these same hostname and port values.

`protocol``://`*`host`*`:`*`port`*`/zone/`*`ZoneId`*

Consider the following code in an agent's **initialize** method:

```
// Set the messaging mode to Push
AgentProperties props = getProperties();
props.setMessagingMode( AgentProperties.PUSH_MODE );

// Set up the HTTP properties
HttpProperties http = getDefaultHttpProperties();
http.setHost( "agent.edustructures.com" );
http.setPort( 12345 );

// Connect to a zone
Zone testZone = getZoneFactory().getInstance(
       "Test", "http://zis:7080/Test");
testZone.connect( ADKFlags.PROV_REGISTER );
```

When connecting to the zone, the ADK would bind to the port 12345 on the local machine at address "agent.edustructures.com". The SIF_Register/SIF_Protocol/SIF_URL element sent to the zone would be formatted as follows:

```
<SIF_URL>http://agent.edustructures.com:12345/zone/Test</SIF_URL>
```

In some cases it is necessary to specify a different SIF_URL hostname and port than those used to bind the local socket. With the 1.5 version of the ADK you can achieve this by calling two new methods of the HttpTransport class: `setPushHost` and `setPushPort`.

For example,

```
HttpProperties http = getDefaultHttpProperties();
http.setHost( “localhost” );
http.setPushHost( “agents01.edustructures.com” );
http.setPort( 12345 );
http.setPushPort( 12345 );
```

This code would cause the agent to bind on the local address “localhost:12345”, but would send a SIF_URL value with “agents01.edustructures.com:12345” as the host-name and port the zone integration server will use to contact the agent.

```
<SIF_URL>http://agent01.edustructures.com:12345/zone/Test</SIF_URL>
```

13. Appendix B Known Issues

Docs and Examples

The Developer Documentation and Example agents are still in a preliminary state. However, there are a few examples agents that are part of this release and the javadoc is up to date.

Issues with SIF 2.x Support

Features Not Yet Implemented

The following SIF 2.0 features are not yet supported in the ADK.

- SIF_ExtendedQuery
- The XML data type (used in <SIF_ExtendedElement> and other elements that support embedded XML)

Issues with SIF 1.x support

The OpenADK can parse and write objects in both SIF 1.x and 2.x formats. However, at this time, there are a number of objects that have known issues in supporting their 1.x format. Here is the list, by package name, of classes that have incomplete support for SIF 1.x

openadk.library.food package

FoodServiceReimbursementRates

openadk.library.hrfin package

Purchasing, FinancialIncomeStatement

openadk.library.instr package

Assignment, CurriculumStructure

openadk.library.library package

LibraryPatronStatus

openadk.library.trans package

StudentTransportInfo

Objects Not Supported

The ADK does not support a small number of objects from the SIF 1.5r1 specification. These objects have been completely redesigned in the SIF 2.0 and are not expected to be widely used in a SIF 1.5r1 environment.

The list of SIF 1.5 objects not supported are:

- Assessment
- AssessmentSection
- AssessmentItem
- AssessmentSubTest
- StudentResultSet

14. Appendix C Upgrading from 1.5

The OpenADK has had some minor changes to some of its core APIs to better support SIF 2.0 and Java 5. An agent that was built using the 1.5 ADK will need to have some minor code changes to correct compiler errors before it will compile against the 2.0 edition of the ADK. This section highlights many of the code changes that may need to be done to agents written using prior versions of the ADK.

Changes to SIF Versions supported

The 2.0 ADK has removed support for the 1.0r1 and 1.0r2 versions of SIF. This edition of the ADK fully all previous certified versions of SIF, which include SIF 1.1 and SIF 1.5r1. The ADK also supports SIF 2.0 and all 2.x versions of SIF.

Agent Provisioning Changes

ADK Initialization

Agents that call ADK.initialize() with a specific SIFVersion and set of SDO libraries, will need to change the second argument. The constants identifying each available SDO library have moved from the SDO class to the SIFDTD class. The SDO class has also been removed from the ADK and will need to be removed if it has been explicitly imported. Here is an example.

```
// ADK 1.5 code
ADK.initialize( ADKExamples.Version, SDO.STUDENT );

// ADK 2.0 code
ADK.initialize( ADKExamples.Version, SIFDTD.SDO_STUDENT );
```

SIFDTD ElementDef constants

In the 1.5 version of the ADK, the SIFDTD class contained a const field that held the ElementDef for each Element or Attribute in the SIF Specification. These constants have been moved to DTD classes within each package. For example, if you referenced SIF-DTD.STUDENTPERSONAL in your code, the corresponding constant is now at Student-DTD.STUDENTPERSONAL.

Provisioning Changes

In the 1.5 version of the ADK, the provisioning APIS, such as zone.setSubscriber() and zone.setProvider(), accepted three arguments. The third argument was a constant from the ADKFlags class that further defined the provisioning options. In the 2.0 version of the ADK, the second argument has changed to accept a specific ProvisioningOptions instance, such as PublishingOptions.

Here is an example.

Example: zone.setPublisher()

```
//
// ADK 1.5 code
//

zone.setPublisher( this,
                SIFDTD.STUDENTPERSONAL,
                ADKFlags.PROV_PROVIDE );

//
// ADK 2.0 code
//

zone.setPublisher( this,
                StudentDTD.STUDENTPERSONAL,
                new PublishingOptions( true ) );
```

Example: zone.setQueryResults()

```
//
// ADK 1.5 code
//

zone.setQueryResults( this,
                SIFDTD.STUDENTPERSONAL );

//
// ADK 2.0 code
//

zone. setQueryResults ( this,
                StudentDTD.STUDENTPERSONAL,
                new QueryResultsOptions() );
```

Message Processing Changes

Changes to The SIFMessageInfo class

The SIFMessageInfo class Encapsulates information about a SIF_Message. In SIF 2.0, a few changes have been made to SIF_Request and SIF_Response messages that have caused changes to the SIFMessageInfo class.

- getSIFRequestVersion() has been changed to getLatestSIFRequestVersion(). This method examines the list of SIF_Versions requested and returns the latest SIFVersion supported by the agent, according to it's current provisioning settings. To get the complete list of all SIF_Versions requested, call getSIFRequestVersions().

Changes to the Event class

ADK 2.0 has created an Enum called EventAction, which has values for ADD, CHANGE, and DELETE. The corresponding ADD, CHANGE, and DELETE constants have been removed from the Event class.

New Enums for Query operators

The Condition class has been removed from the ADK and is now represented as two separate Enum classes. A new Enum, GroupOperators, has been created to represent the And, Or, and None values associated with a SIF_Condition. Another Enum, ComparisonOperators, has been created to represent the values that can be present in a SIF_Operator element. All of the APIs that use these values have been updated to expect the associated Enum constant, rather than an int from the Condition class.

Changes to Mappings

ADK 2.0 has made changes to the Mappings class to allow for more sophisticated mappings strategies to be implemented. The Mappings.map() methods that used to accept an instance of a class implementing the Map interface have been changed to instead require a FieldAdaptor instance. The ADK includes a FieldAdaptor class that uses a Map and is compatible with how the Mappings class worked in ADK 1.5 This class is called the StringMapAdaptor. Here is an example.

```
//
// ADK 1.5 code
//

mappings.map( hashMap, datObject );

//
// ADK 2.0 code
//

StringMapAdaptor sma = new StringMapAdaptor( hashMap );
mappings.map( sma, datObject );
```

SIF Data Object Changes

The SIF Data Object (SDO) classes that represent Objects and Elements in the SIF data model have been changed to better represent the SIF 2.0 strongly-typed data model. In SIF 2.0, some fields have been changed to use XSD datatypes, which are strongly-typed representations of data. The ADK has also updated all of its APIs to match the strongly-typed representation of that data.

For example, the StudentSchoolEnrollment object has a getEntryDate() method. In the 1.5 ADK, this method returned a string, which represented the entry date of the student as a date formatted as “yyyyMMdd”. The 2.0 ADK represents this value as a strongly-typed value and getEntryDate() returns a Calendar instance.

Using Dates in the ADK

The 2.0 ADK has implemented support for dates consistently by using the Java Calendar object for all APIs that represent a date. For example, the SIFMessageInfo.getTimestamp() method, which returns the value of the SIF_Timestamp element, now returns a Calendar object. All APIs that represent dates are implemented using Calendar objects.

The SIFDate class was used in the 1.5 ADK to both represent a date and convert a date to and from a string. However the pattern in the 2.0 ADK is to represent all dates as Calendar instances. Conversion to and from a string, where necessary, is handled by instances of SIFFormatter for each version of SIF supported by the ADK. Here are some examples of differences in date conversions between the 1.5 and 2.0 editions of the ADK.

Retrieving a date from a SIF Object

```
//
// ADK 1.5: Retrieve EntryDate as a Date
//   in ADK 1.5, StudentSchoolEnrollment.getEntryDate()
//   returned a SIFDate
//

SIFDate entryDate = studentSchoolEnrollment.getEntryDate();
Date date = entryDate.toDate();

//
// ADK 2.0: Retrieve EntryDate as a Date
//   in ADK 2.x, StudentSchoolEnrollment.getEntryDate()
//   returns a Calendar
//

Calendar entryDate = studentSchoolEnrollment.getEntryDate();
Date date = entryDate.getTime();
```

Setting a date to a SIF Object from a string representation

```
//
// ADK 1.5: Set EntryDate from a String
//

String entryDate = "19960901";
SIFDate sifDate = new SIFDate( entryDate );
studentSchoolEnrollment.setEntryDate( sifDate );

//
// ADK 2.0: Set EntryDate from a String
//

String entryDate = "1996-09-01";
SIFFormatter formatter = ADK.DTD().getFormatter(SIFVersion.SIF20);
Calendar calEntry = formatter.toDate( entryDate );
studentSchoolEnrollment.setEntryDate( calEntry );
```

Converting a date from its SIF format to a String

```
//
// ADK 1.5: Retrieve EntryDate as a String
//

SIFDate entryDate = studentSchoolEnrollment.getEntryDate();
// SIFDate.toString() returns a string in the SIF 1.x format,
// "yyyyMMdd"
String date = entryDate.toString();

//
// ADK 2.0: Retrieve EntryDate as a String
//

StudentSchoolEnrollment studentSchoolEnrollment;
Calendar entryDate = studentSchoolEnrollment.getEntryDate();

// Retrieve the date in SIF 1.x format "yyyyMMDD"
SIFFormatter formatter = ADK.DTD().getFormatter(SIFVersion.SIF11);
String date = formatter.toDateString( entryDate );

// Retrieve the date in SIF 2.x format "yyyyMMDD"
formatter = ADK.DTD().getFormatter(SIFVersion.SIF20);
date = formatter.toDateString( entryDate );
```

Index