Oracle® Internet File System™

Developer’s Guide

Release 1.1

September 2000

Part No. A75172-04

Oracle Internet File System Developer’s Guide, Release 1.1

Part No. A75172-04

Release 1.1

Primary Author: Dennise Brown

Contributors: Matthew Brandabur, Dennis Dawson, Francine Hyman, Vasant Kumar, Dave Long, Larry Matter, Sylvia Perez, Josh Sacks, Alison Stokes, Ed Yu

Contents

Send Us Your Comments

Developer's Guide, Release 1.1

Part No. A75172-04

Oracle Corporation welcomes your comments and suggestions on the quality and usefulness of this publication. Your input is an important part of the information used for revision.

- Did you find any errors?
- Is the information clearly presented?
- Do you need more information? If so, where?
- Are the examples correct? Do you need more examples?
- What features did you like most about this manual?

If you find any errors or have any other suggestions for improvement, please indicate the chapter, section, and page number (if available). You can send comments to us in the following ways:

- E-mail: ifsdocteam@us.oracle.com
- FAX - 650.605.7104. Attn: Documentation Manager for Product Name
- Postal service:
  Oracle Corporation
  Product Name, Attn: Documentation Manager
  500 Oracle Parkway, Mailstop 5op4
  Redwood Shores, CA 94065
  USA

If you would like a reply, please give your name, address, and telephone number below.

If you have problems with the software, please contact your local Oracle World Wide Support Center.

Preface

The topics covered in this preface include:

Intended Audience

The Oracle Internet File System Developer's Guide is intended for application developers who create custom file system applications using XML and Java.

Structure of the Developer's Guide

The Oracle Internet File System Developer's Guide contains ten chapters and one appendix:

Chapter 1	Getting Started	Introduces the Oracle Internet File System and provides an overview of customization in Oracle *i*FS.
Chapter 2	API Overview	Describes the most commonly used classes of the Oracle *i*FS Java API.
Chapter 3	Working with Documents	Describes how to get started using the Oracle *i*FS API for basic functionality, such as connecting to the database and creating documents.
Chapter 4	Creating Custom Classes	Describes how to create a custom class using XML.
Chapter 5	Using Parsers	Describes how parsing works in Oracle *i*FS and how to write a custom parser.
Chapter 6	Using Renderers	Describes how rendering works in Oracle *i*FS and how to write a custom renderer.
Chapter 7	Using JSPs	Describes how to display an Oracle *i*FS document on the Web using a Java Server Page (JSP).
Chapter 8	Using Agents	Describes how to create a custom program (agent) to perform a specific task.
Chapter 9	Using Overrides	Describes how to use an override to alter server-side functionality.
Chapter 10	Sending E-mail Programmatically	Describes how to send e-mail from an application program.
Appendix A	Error Messages	Describes the most common error messages.

Notation Conventions

The following conventions are used in this manual:

Convention	Meaning
. . .	Vertical ellipsis points in an example mean that information not directly related to the example has been omitted.
`...`	Horizontal ellipsis points in statements or commands mean that parts of the statement or command not directly related to the example have been omitted
`monospaced text`	Monospaced text is used for filenames, pathnames, and code samples.

Related Documents

For more information on Oracle Internet File System, see the following manuals that are included with Oracle Internet File System:

- *Oracle Internet File System Quick Tour*
- *Oracle Internet File System Installation Guide*
- *Oracle Internet File System Setup and Administration Guide*
- *Oracle Internet File System User's Guide*

For more information on the Oracle8*i* database, see the documentation for Oracle8*i*.

The following table lists additional developer documentation available in the Documentation section of the Oracle *i*FS listing on OTN (Oracle Technology Network).

Oracle Internet File System Javadoc	Describes the packages, classes, and methods of the Oracle *i*FS API.
Oracle Internet File System Class Reference	Provides a listing of the class hierarchy and describes the attributes of the Java classes.
Oracle Internet File System XML Reference	Defines and describes the attributes that may be set when creating objects using XML.

1 Getting Started

This chapter covers the following topics:

- Introducing the Oracle Internet File System
- Oracle iFS System
- Application Development Tools
- Customization in Oracle iFS
- Overview of Application Tasks
- Task Reference

Introducing the Oracle Internet File System

The Oracle Internet File System (Oracle *i*FS) is a file system in a database. From the user perspective, Oracle *i*FS looks exactly like any other networked drive on your file system. However, because Oracle *i*FS actually stores documents in a relational database, rather than on a local hard drive, users can perform many tasks using Oracle *i*FS that are not possible using standard file systems.

For example:

- Files placed into Oracle *i*FS can be automatically indexed, contents and all, for complex searching.
- Complex security models can be implemented with ease: individual or group access can be granted to files or directories using simple commands.
- The full functionality of Oracle *i*FS is available through an array of interfaces already familiar to most users.
- Oracle *i*FS supports an extendible list of 150 different file types, and includes robust support for sophisticate content management, with or without user customization.

Oracle *i*FS Advantages for Developers

For developers, using Oracle *i*FS offers several specific advantages:

- Storing all data, including both files and relational data, in a single system simplifies application development. Rather than having to coordinate two separate data stores, you now only need to keep track of a single repository.
- You can customize the behavior and appearance of Oracle *i*FS using standard coding tools: XML, HTML, and Java.
- The Oracle *i*FS system provides facilities to speed development of custom file systems, Web-based applications and content management applications.

Oracle *i*FS System

As a developer of Oracle *i*FS applications, you have access to all the components of Oracle *i*FS:

- The Oracle *i*FS Repository
- The Oracle *i*FS Client Software
- The Oracle *i*FS Protocol Servers
- An Extensible Document Hierarchy
- A Java-based API

The Oracle *i*FS Repository

The Oracle *i*FS repository provides a single storage facility for all of your files, whether they are a standard type, such as XML, or a custom type that you define. This single storage facility means that files are managed consistently, regardless of the protocol used to manipulate them.

The Oracle *i*FS Client Software

You can use standard clients, such as Windows Explorer; out-of-the-box Oracle *i*FS clients, such as the Web interface: You can also create a custom client to access Oracle *i*FS data. TCP/IP is used for communication between clients and the Oracle *i*FS server.

The Oracle *i*FS Protocol Servers

Out-of-the-box, Oracle *i*FS ships with a set of standard protocol servers: SMB, HTTP, FTP, SMTP, and IMAP4. Each Oracle *i*FS protocol server accepts commands from a standard client and maps those commands to repository operations.

An Extensible Document Hierarchy

The Oracle *i*FS document hierarchy may fit your application needs as is. If the out-of-the-box hierarchy fits your application needs only in part, you can easily create custom document classes using XML to define custom attributes and Java to implement custom processing.

A Java-based API

The Oracle *i*FS Java API consists of a set of classes and methods that custom applications can use to access the repository and perform file management operations.

Application Development Tools

To use Oracle *i*FS in custom application development, you need a few readily available tools:

- A Java Integrated Development Environment (IDE)
- An HTML Editor
- An XML Editor

Which Tool to Use?

The following table lists common customization tasks and the corresponding tool to use.

Task	**Tool**
Create a custom type (class).	XML Editor
Modify a custom type (class).	Oracle *i*FS Manager
Create a custom Java Bean.	JDeveloper or other Java IDE
Create a JSP.	HTML editor, JDeveloper or other Java IDE
Display information using a JSP.	Java Web Server or Apache Web Server
Create a custom parser, renderer, or agent.	JDeveloper or other Java IDE
Register a parser, renderer, or JSP.	Oracle *i*FS Manager or XML Editor
Register an agent.	Oracle *i*FS Server Manager

Customization in Oracle *i*FS

The Oracle Internet File System was built with ease of customization in mind. Depending on the requirements of your application, you can choose from three levels of customization:

- No customization
- Basic customization
- Advanced customization

No Customization

For many applications, the file system management and content management features of Oracle *i*FS mean that no customization is required. Oracle *i*FS provides many out-of-the-box capabilities that you might expect to add with customization to a standard document-centered application, such as:

- Versioning
- Check-in and check-out
- Locking and unlocking

Basic Customization

For applications that require only adding custom attributes to the existing Document class, basic customization can be done using XML, with no Java programming required.

When you use XML to define custom document attributes, you have access to parsing and rendering facilities provided by the SimpleXmlParser and SimpleXmlRenderer included with Oracle *i*FS.

Advanced Customization

For applications with complex requirements, customization can be carried out in Java, starting with the classes provided in the Oracle *i*FS Java API.

With Java programming, you can add the following types of customization:

- Custom agents
- Custom parsers and renderers
- Custom overrides
- JSPs

Overview of Application Tasks

The following table lists common application tasks and the Oracle *i*FS functionality you might use to accomplish the task.

Application Requirement	Oracle *i*FS Functionality
Make a database connection.	Connect to the repository.
Manage documents and folders.	Create, update, and delete repository objects.
Store files and folders with custom attributes.	Create a custom document type, defining custom attributes or methods.
Extract document components and store them separately.	Write a custom parser.
Change the range, values, or format of a parsed document when it is reconstructed.	Write a custom renderer.
Display dynamic web content based on the file system's contents.	Write a JSP.
Perform a specific task before or after a given database event.	Write an agent. Write an override to modify the default behavior of the repository.

Task Reference

The Oracle *i*FS Java API is a set of classes that allow an application developer to create, update, and delete repository objects. The API classes allow you to perform in a custom manner the same functions provided through the Oracle *i*FS protocols and interfaces. The following table lists these functions and their corresponding reference in this document.

Task	Reference
Become familiar with key classes of the API.	Chapter 2, "API Overview"
Connect to the repository.	Chapter 3, "Working with Documents"
Create, update, and delete repository objects.	Chapter 3, "Working with Documents"
Read and write content.	Chapter 3, "Working with Documents"
Extend the API classes.	Chapter 4, "Creating Custom Classes"
Create and register a custom parser.	Chapter 5, "Using Parsers"
Create and register a custom renderer.	Chapter 6, "Using Renderers"
Create and register a JSP.	Chapter 7, "Using JSPs"
Create and register an agent.	Chapter 8, "Using Agents"
Override the default behavior of the repository.	Chapter 9, "Using Overrides"
Send e-mail programmatically.	Chapter 10, "Sending E-mail Programmatically"

Note: For reasons of conciseness and clarity, the sample code in this Developer’s Guide is presented as code fragments pulled from executable code. For the complete executable version of these files, download the Oracle *i*FS Developer Kit from the Software section of the Oracle *i*FS listing on OTN (Oracle Technology Network).

2 API Overview

This chapter covers the following topics:

Introducing the Oracle *i*FS Java API

The Oracle Internet File System Java API provides classes and methods that allow you to customize all of the functionality included in the out-of-the-box Oracle *i*FS interfaces. These classes include:

- Classes for storing documents and attributes, such as Document and Folder.
- Classes that provide document management functionality, such as check-in/check-out and version control.
- Classes that provide security, such as Users, Groups, and ACLs.

Because the functionality is extensive, the API includes a large number of classes, which are organized into packages for ease of use. Because the large number of classes means that the Javadoc is extensive, this chapter provides a quick introduction to a few of the most frequently used Oracle *i*FS classes.

A good way to begin the process of familiarization with Oracle *i*FS would be to:

- Scan the classes highlighted in this chapter.
- Consult the Javadoc for each of these key classes.

The Oracle *i*FS API Packages

The classes of the Oracle *i*FS API are organized into 25 packages. You can see this complete structure by going to the Oracle *i*FS Javadoc and clicking on the link labelled "Overview." Of these 25 packages, seven contain the classes that are most frequently used for developing Oracle *i*FS custom applications. The following table describes the types of classes included in each of these seven packages.

Package Name	Description
`oracle.ifs.agents.common`	Classes used to create and manage agents.
`oracle.ifs.beans`	The most important package. Contains most of the classes used by application developers.
`oracle.ifs.beans.parsers`	Classes used to create custom parsers.
`oracle.ifs.search`	Classes used for search functionality.
`oracle.ifs.server`	Server-side classes for managing objects in the database.
`oracle.ifs.server.renderers`	Classes used create custom renderers.
`oracle.ifs.common`	Utility classes used by both bean-side and server-side classes.

The LibraryObject Class

The LibraryObject class forms the highest level of the class hierarchy of the `oracle.ifs.beans` package. All classes that represent persistent objects in `oracle.ifs.beans` inherit from LibraryObject, so LibraryObject is the base class for these objects, providing functionality common to all the classes. For example, the methods that provide the following functions are located in the LibraryObject class:

- Updating an object
- Deleting an object
- Setting / getting attributes for an object

LibraryObject has three subclasses, which are abstract superclasses for the objects below them in the hierarchy:

- PublicObject
- SystemObject
- SchemaObject

The following table describes the purpose of the three LibraryObject subclasses.

Class	Purpose	Direct Interaction with End Users?
`PublicObject`	The superclass for all objects that end users deal with directly, such as documents and folders.	Yes
`SystemObject`	The superclass for all system-wide utility classes. These classes are used to help manage PublicObjects.	No
`SchemaObject`	The superclass for classes that manage how all repository information is stored and managed.	No

The LibraryObjectDefinition Class

To create any Oracle *i*FS object is a two-stage process:

1. Build the definition of the object, using the appropriate Definition class, such as DocumentDefinition.
2. Pass the Definition object to the createPublicObject() method (or its analogue in other classes, createSystemObject() or createSchemaObject()).

All Definition classes inherit from LibraryObjectDefinition and PublicObjectDefinition, as shown in the following class hierarchy:

```
java.lang.Object
  +--oracle.ifs.beans.LibraryObjectDefinition
        +--oracle.ifs.beans.PublicObjectDefinition
               +--oracle.ifs.beans.DocumentDefinition
```

The oracle.ifs.beans Class Hierarchy

You can use the classes of the `oracle.ifs.beans` package as-is, if they meet your requirements. You can also subclass these classes to create custom classes. Deciding which class you want to subclass is a key decision, because it determines the functionality your custom class inherits. To make that decision, you need to become familiar with:

- The class hierarchy of the `oracle.ifs.beans` package, which contains the most frequently used classes.
- The most commonly used individual classes and their unique attributes.

Most of the application development work you do will use the classes that make up the three subclasses of LibraryObject. Before you begin working with these classes, you may find it useful to review this abbreviated class hierarchy, which includes the three LibraryObject subclasses:

- PublicObject
- SystemObject
- SchemaObject

To provide a starting point, several of the most commonly used classes are described later in this chapter. All of the classes are described in the Javadoc.

```
PublicObject
    AccessControlList
        SystemAccessControlList
        ClassAccessControlList
    ApplicationObject
        ContentQuota
        PropertyBundle
            PolicyPropertyBundle
            ValueDefaultPropertyBundle
            ValueDomainPropertyBundle
            ServerDetail
            ServerRequest
        ServerSubClass
    Category
```

```
        MountPoint
    DirectoryObject
        DirectoryGroup
            AdministrationGroup
        DirectoryUser
    Document
        MailDocument
    Family
    Folder
        Mailbox
        Message
        MailFolder
    SearchObject
    SelectorObject
    Template
    UserProfile
        PrimaryUserProfile
        ExtendedUserProfile
            EmailExtendedUserProfile
    VersionSeries
    VersionDescription
```

SystemObject

```
    AccessControlEntry
    AuditEntry
    AuditRule
    ContentObject
    ExtendedPermission
    Format
    Media
        MediaFile
        MediaLob
            MediaBlob
        MediaReference
    PermissionBundle
    Policy
    Property
    Relationship
        BranchRelationship
        FolderRelationship
            FolderPathRelationship
                BodyPartPathRelationship
        GroupMemberRelationship
        NamedRelationship
```

```
SchemaObject
    Attribute
    ClassDomain
    ClassObject
    ValueDomain
    ValueDefault
```

Three groups of special classes are closely related to corresponding PublicObject classes, so they are not included in this list:

- Definition classes are used to group attributes before creating a PublicObject.
- Tie classes are used to add behavior to all subclasses of a specific PublicObject.
- Server classes (preceded with an S_) are used to work with objects inside the Oracle *i*FS respository.

This list of classes is also abbreviated to include only the more commonly used classes. For a complete Oracle *i*FS class hierarchy, see the Oracle Internet File System Class Reference, which provides a listing of the class hierarchy and describes the attributes of the Java classes. Access the Class Reference in the Documentation section of the Oracle *i*FS listing on OTN (Oracle Technology Network).

The PublicObject Class

Of all the classes in the Oracle *i*FS API, PublicObject is the most significant. PublicObject is the abstract superclass for all user-related classes and thus defines the attributes and methods that are common to all of these classes. Oracle *i*FS maintains all of the attributes defined by PublicObject automatically. For example, Oracle *i*FS will ensure that each file has a Name attribute. For a list of these attributes, see"Public Object Attributes".

These attributes are important because you can use them, as-is, in your custom type definition files. That means that the names of these pre-defined attributes are Oracle *i*FS keywords, so you cannot use them for custom attributes. For example, Name and Description are PublicObject attributes, so if you need to define a custom class that has a similar attribute, you must call that attribute something other than Name or Description, such as ApproverName or DocumentDescription.

Because PublicObject is an abstract class, it is never directly instantiated. Rather, the user-related classes are subclasses of PublicObject. Two general types of user-related classes are subclassed from PublicObject:

- Classes that end users deal with directly, such as Document and Folder objects.
- Classes that play a supporting role, such as Groups and Access Control Lists.

Characteristics of Public Objects

Because the repository is based on a single inheritance tree, each class inherits attributes and methods from PublicObject. Thus, all classes that inherit from PublicObject share some common characteristics:

- Public objects are the only objects that appear in folders.
  - Foldering is not required: A public object can exist as an unfoldered object.
  - In addition, a public object can appear in multiple folders.
- Public objects are access-controlled. Methods implemented on this class can be used to grant, revoke, set, and inquire on permissions.
- Public objects have a large set of attributes, some of which you must set and others that are set automatically:
  - Some attributes are system-set, such as the CreateDate and LastModifyDate attributes.
  - Other attributes can be set from the API, such as the Owner, Description, and ACL (Access Control List) attributes.
- Public objects are the only objects that can be versioned.

These subclasses inherit all the attributes of the PublicObject class, as well as a set of attributes specific to the subclass. For example, the Document class has two sets of attributes:

- Inherited: The Document class inherits a set of common attributes from the PublicObject class, such as Owner and CreateDate.
- Class-specific: The Document class also includes attributes specific to the Document class, such as ContentObject and ReadByOwner.

Public Object Attributes

The following table lists the attributes of the PublicObject class. Use this list for two purposes:

- To find out if there is an attribute defined "out-of-the-box" that will meet your needs.
- To choose a unique name for any custom attributes. Because attribute names must be unique within Oracle *i*FS, names of all pre-defined attributes of classes in the Oracle *i*FS API are reserved words, and may not be used to identify custom attributes.

Attribute	Datatype	Comments
`Name`	String (700)	Name of the document. Required.
`Description`	String (2000)	Detailed description of the document.
`Owner`	DirectoryObject	Owner of the document. Set by system. (Can also be set explicitly.)
`ACL`	PublicObject	ACL assigned to this document.
`Family`	PublicObject	The Family related to this document. Used in connection with versioned document. For non-versioned documents, value is NULL.
`ResolvedPublicObject`	PublicObject	Used by non-document objects. Not relevant to documents.
`CreateDate`	Date	Date the document was created.
`Creator`	DirectoryObject	User who created the document.
`LastModifyDate`	Date	Date the document was modified.
`LastModifier`	DirectoryObject	User who modified the document.
`Deletor`	PublicObject	Reserved for future use.
`PolicyBundle`	PublicObject	Can be used to control any behavior overrides for this object. Used to map the desired renderer.
`PropertyBundle`	PublicObject	Used for storing ad hoc name/value pairs.
`SecuringPublicObject`	PublicObject	Not relevant for documents. Indicates that security for this object is based on the ACL for another object.

Attribute	Datatype	Comments
`ExpirationDate`	Date	Date the document will be deleted.
`LockState`	Integer	Indicates that document has some type of lock. Locks can be set on a permanent or session basis.
`Flags`	Integer	Reserved for system use. Used by protocol servers to store miscellaneous status bits.
`LockedForSession`	Long	Indicates that the document is locked for the current session. Attribute holds the session identifier.
`ID*`	Long	Unique numeric identifier for current object. Set by system.
`ClassID*`	Long	Unique numeric identifier for class of current object. Set by system.

*An intrinsic identifier, not a PublicObject attribute. Included in this list because this identifier name is also a reserved word, and may not be used for custom attributes.

Do You Need to Create a Custom Oracle *i*FS Document?

Assuming that your goal as a developer is to create custom applications quickly, the first question to consider is, "Will a standard Oracle *i*FS document meet the needs of my application?"

The answer depends on whether the attributes of your document are adequately described in the list of standard attributes defined for the Oracle *i*FS Document class.

To decide whether your document is a "standard Oracle *i*FS document," consult the list of "Public Object Attributes". If your document can be defined using only these standard attributes:

- It is a standard Oracle *i*FS document.
- You do not need to define a custom document class to store this document in Oracle *i*FS.

If your document has custom attributes that need to be specifically defined, see Chapter 4, "Creating Custom Classes".

In special circumstances, you may need to create a custom subclass even though your application does not require defining any custom attributes. For example, if your application requires identifying all files of a certain type, such as Word files or

HTML files, and then manipulating the file contents or attributes in specific way, you would need to define that type as a custom class.

User-related Classes

Another way to quickly familiarize yourself with some of the key classes of the Oracle *i*FS Java API is to consider the key user-related classes by function. The following table lists the key classes that belong to each group and the purpose of the group. (This table includes some classes that are not described in this chapter. For more information about these classes, consult the Javadoc.)

Function	Key Classes	Purpose
Hold Content	Document	Holds content data.
	DocumentDefinition	Used to build a document in memory.
		Note that all classes used to create objects, such Folder and Family, also have corresponding Definition classes.
	ContentObject	Holds a reference to the actual document content.
Organization	Folder	Groups objects together.
Versioning	Family	Groups a series of related versioned public objects.
	VersionSeries	Groups a series of related versioned public objects within a family. A family can have more than one series.
	VersionDescription	Creates an individual versioned public object within a VersionSeries object.
Security	AccessControlList (and subclasses)	Contains a list of security entries called Access Control Entries (ACEs) that grant or revoke privileges on the object to a user or group.
	AccessControlEntry	An entry in an Access Control List (ACL), specifying access privileges for a single user or group.
Support Objects	ApplicationObject	Acts as superclass for custom objects.

Function	Key Classes	Purpose
	PropertyBundle	Used for managing custom lists of name/value pairs.

Javadoc for User-Related Classes

To understand a given user-related class, consult the Javadoc for these three classes related to whichever class you choose:

- Start with the `PublicObject` class, because it is the abstract class from which all user-related classes inherit.
- Then check the subclass of PublicObject that you are interested in, such as the `Document` class.
- To familiarize yourself with the attributes of the class, see the related Definition class, such as the `DocumentDefinition` class.

Document and Folder Classes

Four classes are used to work with folders, documents (files), and their contents:

- The Document Class
- The DocumentDefinition Class
- The ContentObject Class
- The Folder Class

If your application deals with a document (file), you will need to create three objects:

- A Document object to store the attributes and a reference to the ContentObject in the repository.
- A DocumentDefinition object to hold the attributes of the file. The DocumentDefinition object is transitory and is used to create the Document object.
- A ContentObject to store the actual content of a Document object.

In addition, if you want to place the file in a folder, you will need to create a Folder object.

The Document Class

Package Name: oracle.ifs.beans

Class Name: Document

ParentClass: PublicObject

Purpose: Adds the ability to store and manage the content of a file.

Methods: The Document class provides a number of methods specific to getting and setting the associated ContentObject.

Uses: Use the Document class to store a document or file.

Attributes: The following attributes are unique to the Document class:

Attribute	Datatype	Description
`ContentObject`	ContentObject	The ObjectID of the ContentObject that holds the content for this document.
`ReadByOwner`	Boolean	A flag that indicates whether the document owner has read the current content. Used by IMAP to indicate whether the text of a message has been read.

The DocumentDefinition Class

Package Name: oracle.ifs.beans

Class Name: DocumentDefinition

ParentClass: PublicObjectDefinition

Purpose: The DocumentDefinition class is used to construct a Document object. This subclass of PublicObjectDefinition sets the default ClassObject to "DOCUMENT". An instance of a DocumentDefinition is passed to LibrarySession.createDocument() to actually construct the new document.

Methods: The DocumentDefinition class provides a number of methods specific to storing and manipulating content.

Uses: Use the DocumentDefinition class to create a transient, in-memory object to hold the attributes of a Document object while you are creating it. Once the Document Definition is passed to the createDocument() method, the definition object is discarded.

Attributes: The attributes for the DocumentDefinition class are the same as those for the Document class.

The ContentObject Class

Package Name: oracle.ifs.beans

Class Name: ContentObject

ParentClass: SystemObject

Purpose: Stores the actual content of a document.

Methods: The ContentObject class provides a number of methods specific to getting and setting its attributes, including content, content size, and format.

Attributes: The following attributes are unique to the ContentObject class:

Attribute	Datatype	Description
`CharacterSet`	String	Character set used to represent the language for this ContentObject.
`Content`	Long	A pointer to the actual content of the document.
`ContentSize`	Long	Size of content in bytes.
`Format`	SystemObject	Format/MIME type for this ContentObject.
`Language`	String	Language for this ContentObject.

The Folder Class

Package Name: oracle.ifs.beans

Class Name: Folder

ParentClass: PublicObject

Purpose: Adds the ability to manage references to other files.

Methods: The Folder class provides methods to manipulate items in a folder, such as getItems().

Uses: Create a custom subclass of the Folder class if you need an object that manages references to other objects.

Attributes: The Folder class does not contain any unique attributes.

Security Classes

Security in Oracle *i*FS is provided by AccessControlLists (ACLs), which provide security information for a specific object through a collection of AccessControlEntries (ACEs). Each ACE grants or revokes specific permissions related to that object toa user or group.

The AccessControlList Class

Package Name: oracle.ifs.beans

Class Name: AccessControlList

ParentClass: PublicObject

Purpose: Used to specify access rights to documents.

Methods: The AccessControlList class provides specific methods to manipulate AccessControlEntries.

Uses: Use the AccessControlList class to acquire information about a group of AccessControlEntries.

Attributes: The following attributes are unique to the AccessControlList class:

Attribute	Datatype	Description
`Shared`	Boolean	Indicates whether this ACL is shared. Default value is TRUE.

The AccessControlEntry Class

Package Name: oracle.ifs.beans

Class Name: AccessControlEntry

ParentClass: SystemObject

Purpose: Used to specify access rights for a specific user or group.

Methods: The AccessControlEntry class provides specific methods to obtain information from AccessControlEntries.

Uses: Use the AccessControlEntry class to acquire information about a specific AccessControlEntry.

Attributes: The following attributes are unique to the AccessControlEntry class:

Attribute	Datatype	Description
`AccessLevel`	Long	Required.
`Acl`	PublicObject	ACL with which this ACE is associated. Required.
`ExtendedPermissions`	SystemObject Array	Optional.
`Granted`	Boolean	Indicates whether this grantee has this permission. Default is TRUE. Required.
`Grantee`	DirectoryObject	DirectoryUser object for person or group this permission is for. Required.
`PermissionBundles`	SystemObject Array	Optional.
`SortSequence`	Long	Required.

Session Classes

If a user wants to add data to or retrieve data from the Oracle *i*FS repository, that user must have a working session to create the connection to the repository. Creating this session is the first task of every Oracle *i*FS application, and requires the following classes:

- The LibraryService Class
- The LibrarySession Class

The LibrarySession class is a lynch-pin class in Oracle *i*FS. All persistent objects, that is, objects stored in the Oracle *i*FS repository, are created from the LibrarySession class.

The LibraryService Class

Package Name: oracle.ifs.beans

Class Name: LibraryService

ParentClass: java.lang.Object

Purpose: The LibraryService class is used for connecting to the Oracle *i*FS server and launching sessions via the LibrarySession object.

Methods: The LibraryService class is a factory class for LibrarySession. The LibraryService class has a public constructor and one method, the connect() method. The connect() method either returns a LibrarySession or throws an exception indicating a connect failure.

Uses: Use the LibraryService class to create an instance of LibrarySession.

The LibrarySession Class

Package Name: oracle.ifs.beans

Class Name: LibrarySession

ParentClass: java.lang.Object

Purpose: Each instance of LibrarySession represents an authenticated user session, that is, a repository connection.

Methods: The LibrarySession class provides the key methods used to create and free repository objects.

Uses: Use the LibrarySession class to manipulate all persistent objects.

Tie Classes

Tie classes allow you to alter the out-of-the-box behavior of the Oracle *i*FS classes by "tie-ing" into the hierarchy at any level. Each Tie class provides an implementation class for its corresponding Oracle *i*FS class. Thus, the implementation class for Document is TieDocument. Because Tie classes provide the implementation classes for Oracle *i*FS classes, the way to alter the behavior of Document and all its subclasses is to extend TieDocument. Tie classes allow you to insert changed behavior in a single place, which leads to cleaner code and ease of maintenance.

Tie classes are "empty" classes that hold a place in the Oracle *i*FS hierarchy so you can customize the behavior of existing Oracle *i*FS classes and have the new behavior become part of the inheritance structure. When you write Java code to change the functionality of the Document class, your new class extends not Document, but TieDocument.

To understand Tie classes, you need to know that the Tie classes come below their corresponding classes in the Oracle *i*FS class hierarchy. For example, consider the following hierarchy for TieDocument:

```
java.lang.Object
  +--oracle.ifs.beans.LibraryObject
            +--oracle.ifs.beans.TieLibraryObject
```

```
          +--oracle.ifs.beans.PublicObject
            +--oracle.ifs.beans.TiePublicObject
                  +--oracle.ifs.beans.Document
                                 +--oracle.ifs.beans.TieDocument
```

Suppose that you want to subclass the Document class, creating three new classes: Reports, Specs, and Memos. Assume that there is common functionality you want to include in Document itself, so that all three new classes, as well as all future subclasses of Document, will inherit that behavior. This common functionality could be that the Description field should always contain the company name followed by name of the user who is creating or modifying the document. One approach would be to place the code for the common Description functionality in each of the new subclasses. However, this would mean the code would be in three places (and possibly more, in the future), which would add complexity and increase future maintenance requirements.

For clarity and ease of maintenance, it would be preferable to alter the behavior of the base Document class, so that all three new subclasses, as well as any future subclasses, would inherit the behavior. However, because the Document class is part of the Oracle *i*FS API, you cannot extend it directly. Tie classes are provided for this purpose. Tie classes provide the implementation classes for the Oracle *i*FS classes, so the way to alter the behavior of Document and all its subclasses is to extend TieDocument.

Returning to our example, the way to achieve the preferred implementation is to add the new Description functionality in one place: the TieDocument class. Then, if Reports, Specs, and Memos all subclass TieDocument, they will automatically inherit the new behavior.   For an example of a class that extends TieDocument, see "Sample Code: Create an Instance Class Bean", in Chapter 4, "Creating Custom Classes".

Tie classes are provided for both bean-side and server-side classes, so there is both a TieDocument and an S_TieDocument class.

Tie classes are only relevant to existing Oracle *i*FS classes. When you extend your own custom classes, you have access to your original code, and can extend the base class. For example, to create a custom StatusReports class that inherits functionality from your custom class Reports, simply extend Reports.

Note: If you place custom code directly in TieDocuments, be sure to alter your CLASSPATH so the reference to the customized class precedes the reference to the repository .jar file.

Server Classes

In the Oracle *i*FS Java class hierarchy each Oracle *i*FS object has two representations:

- The bean-side representation, known by the object name, such as "Document." The bean-side classes are stored in the `oracle.ifs.beans` package.
- The server-side representation, known by the object name preceded with "S_", such as "S_Document." The server-side classes are stored in the `oracle.ifs.server` package.

Most application work is done with the bean-side classes. However, two types of customization require working with objects in the database and need to use the server-side classes:

- Renderers
- Overrides

Until you work with renderers or overrides, you only need to be concerned with the bean-side classes. Behind the scenes, however, Oracle *i*FS uses the S_ classes, such as S_Document, for processing that must be carried out on the document in the database. When you encounter S_ classes in the sample code, you know that you are dealing with the server-side representation of the object.

For more information on using server-side classes for specific types of customization, see:

- Chapter 6, "Using Renderers", "Using Server-Side Classes with Renderers".
- Chapter 9, "Using Overrides", "Attributes and Server-Side Classes".

3 Working with Documents

This chapter covers the following topics:

How Documents Are Stored in the Repository

The Internet File System repository provides persistent storage for all of your documents (files) in database tables. Documents are stored in the repository in two parts:

- Document *attributes* are stored in one database table.

  The attributes are stored in their appropriate datatypes as a set of columns in that database table.

- Document *content* is stored in a separate table.

  The content table stores the physical byte stream for document content as a LOB.

Documents and Folders

In Oracle *i*FS, documents are stored independently from folders. No single database table contains a folder and its associated documents. Conceptually, picture three separate tables:

- A table that stores documents (actually, one table for attributes and one for content)
- A table that stores folders
- A table that stores relationships between documents and folders

Oracle *i*FS uses the "reference model" rather than the "containership model." The "reference model" architecture allows multiple folders to include references to the same document, while only one copy of the document is actually stored. (This approach is also known as "multiple parents.") In the "containership model," if three folders contain the same document, there may be three copies of the document.

Connecting to the Repository

Before a program can perform any function in Oracle *i*FS, the current user must have a working session to provide a connection to the repository. This session allows the application to create and save documents to the repository. Creating this session is the first task of every Oracle *i*FS application.

Connecting to the repository is a two-step process:

1. Create an Instance of LibraryService.
2. Obtain an Instance of LibrarySession.

LibraryService is a factory class for LibrarySession. That is, LibraryService provides a method you can invoke to create a LibrarySession. The LibraryService.connect() method returns an instance of the class oracle.ifs.beans.LibrarySession. Each instance of LibrarySession represents an authenticated user session. The LibrarySession class provides the method used to create repository objects.

Step 1: Create an Instance of LibraryService

To create an instance of LibraryService, use the LibraryService constructor, which expects no parameters.

```
LibraryService ifsService = new LibraryService();
```

Step 2: Obtain an Instance of LibrarySession

To obtain an instance of LibrarySession, use the LibraryService.connect() method. The arguments for the connect() method are a Credential object and a ConnectOptions object, which must be created before invoking connect().

```
CleartextCredential credentials = new CleartextCredential(username, password);
ConnectOptions connectOpts = new ConnectOptions();
connectOpts.setLocale(Locale.getDefault());
connectOpts.setServiceName(servicename);
connectOpts.setServicePassword(servicepassword);

LibrarySession ifs = ifsService.connect(credentials,connectOpts);
```

where:

Parameter Name	Datatype	Description
`credentials`	CleartextCredential	Object containing the user name and password. - `username` is the Oracle *i*FS user name. - `password` is the Oracle *i*FS user password.
`connectOpts`	ConnectOptions	Object containing the locale, service name, and service password. - Locale is the standard Java Locale object. - `servicename` is the name of the Oracle *i*FS service to which the user is being connected. The properties file is located in the package `oracle.ifs.server.properties`. The properties file name must be of the form `Service.properties`. - `servicepassword` is the database password for the owner of the Oracle *i*FS schema.

The connect() Method

The connect() method is overloaded to allow Oracle *i*FS to use other types of credentials to validate Oracle *i*FS users. If the arguments provided to the connect() method are accepted, connect() returns an instance of LibrarySession.

The most common form of the connect() method, shown above, uses the Cleartext credential manager for user authentication.

Javadoc Reference

For more information about connecting to the repository, see the Oracle *i*FS Javadoc:

- `oracle.ifs.beans.LibraryService`
- `oracle.ifs.beans.LibrarySession`
- `oracle.ifs.common.CleartextCredential`
- `oracle.ifs.common.ConnectOptions`

Creating a New Document

Once you have a valid connection, you can create a new document and insert it into the repository.

Creating a document is a two-step process:

1. Create a Document Definition Object.
2. Create a New Document.

The method used to create a new repository object is LibrarySession.createPublicObject. This method expects a single argument, a definition object defining the object that is being created.

A LibraryObjectDefinition object is created using the LibraryObjectDefinition constructor. This constructor expects a single argument, which is a valid LibrarySession object. Instances of LibraryObjectDefinition are transient; they do not map to repository objects. A LibraryObjectDefinition object specifies:

- The class of object being created.
- Initial values for some or all attributes of the new object.

Why Create a Definition First?

In Oracle *i*FS, you create a Definition object prior to creating any PublicObject. The Oracle *i*FS architecture requires that you create the Definition object for reasons of efficiency and speed. The definition object is a transient, in-memory object. Once the object has been created in the repository, the definition object is discarded. By assembling the definition object in its entirety first, before creating the actual document object, you may save many trips to the repository. By creating the definition object, you "build" the object in memory first. Then, when the object is complete, you pass all the attributes in a single database call.

Creating PublicObjects

The PublicObject class is the superclass for all the objects that end users manipulate directly. These objects include both frequently used objects, such as Document and Folder objects, and supporting objects, such as Access Control lists. This topic focuses on creating a Document object by passing a DocumentDefinition to the createPublicObject() method. However, the same process is used for all PublicObjects. For example, to create a folder, you would create a Folder Definition and pass it to the createPublicObject() method.

Create a Document Definition Object

To create a Document Definition object:

1. Create a new instance of DocumentDefinition, passing in the current session.
2. Specify the object's attributes.

Sample: Create a Document Definition Object

```
DocumentDefinition newDocDef = new DocumentDefinition(ifsSession);

newDocDef.setAttribute("NAME", AttributeValue.newAttributeValue
   ("Hello_World.txt"));
newDocDef.setContent("Hello World");
```

For more information about setting attributes, see "Setting Attributes".

Create a New Document

To create a new document (or other PublicObject), call the ifsSession.createPublicObject() method, passing the DocumentDefinition as a parameter.

Sample: Create a Document Object

```
Document doc = (Document) ifsSession.createPublicObject(newDocDef);
```

Note that the return from the createPublicObject() method is cast to the appropriate object type (in this example, a Document object).

The createPublicObject method() method takes one parameter:

Parameter Name	Datatype	Description
`newDocDef`	LibraryObjectDefinition	Definition of the attributes of the object that is being created.

Putting a Document in a Folder

By default, new instances of PublicObject and its subclasses, such as Document, are created unfoldered. If you want users to be able to navigate to an object, the object must be foldered, so it is important to folder new documents as soon as they have been created. (Users can use the Find function to find foldered objects only.)

To add an item to a folder, use the Folder.addItem() method, which takes a single argument: the object to be added to the folder. You also need to specify which folder or folders the object should be added to. Frequently, an application requires that a document be placed in a user's home folder. To do this, you need to access the current user's home folder, using the getHomeFolder() method, which is a three-step process:

1. Use theLibrarySession.getDirectoryUser() method to obtain the DirectoryUser object for the current user.
2. Once you have obtained the DirectoryUser object, pass it to the getPrimaryUserProfile() method.
3. Once you have obtained the PrimaryUserProfile object, invoke the getHomeFolder() method on it.

Sample: Putting a Document in a Folder

```
DirectoryUser thisUser = ifsSession.getDirectoryUser();
PrimaryUserProfile userProfile = ifsSession.getPrimaryUserProfile(thisUser);
Folder homeFolder = userProfile.getHomeFolder();

homeFolder.addItem(doc);
```

Working with Attributes

Working with attributes includes three functions:

- Getting Attributes
- Setting Attributes
- Defining Explicit Getters and Setters

Note that while the standard convention is to call methods that access attribute values "accessor methods" and methods that change attribute values "mutator methods," the more common terms, "getter" and "setter" methods, are used here.

Getting Attributes

In Oracle *i*FS, you can get attribute values in two ways:

- Using an Explicit Getter Method
- Using the Generic getAttribute() Method

Using an Explicit Getter Method

Explicit getter methods are often provided by the definition object to simplify getting attribute values. If the attribute you want to set has an explicit getter method defined for it, using that method is the easiest and most efficient way to get the value of an attribute.

In the following example, an explicit getter method is used to get the value of Dateofbirth.

```
Date dob = getDateofbirth();
```

Using the Generic getAttribute() Method

Generically, the LibraryObject.getAttribute() method is used to get an attribute. The method takes one parameter, the name of the required attribute. This argument should be supplied using the static variable defined in the instance Bean.

The getAttribute() method returns an instance of the class oracle.ifs.common.AttributeValue. The AttributeValue class provides a set of get*DataType*() methods that return the value of the attribute in the required format. To use the get*DataType*() methods, you must pass in a valid LibrarySession.

In the following example, the getString() method is used to get the value of Fullname from the AttributeValue object returned by getAttribute().

```
AttributeValue av = getAttribute(FULLNAME_ATTRIBUTE);
String fullName = av.getString(ifsSession);
```

Setting Attributes

When you create a Definition object, you can specify initial values for some or all attributes of the new object. In Oracle *i*FS, you can set attribute values in two ways:

- Using an Explicit Setter Method
- Using the Generic setAttribute() Method

Using an explicit setter method is the simplest approach. However, explicit setters are provided only for tricky attributes. For all other attributes, you must use the generic approach. In the example of creating a DocumentDefinition object, you saw both ways in action:

- Setting the Name attribute required the generic approach, setAttribute().
- Setting the Content used an explicit setter method, setContent().

Using an Explicit Setter Method

Explicit setters are occasionally provided by the Definition object to simplify setting attribute values. If the attribute you want to set has a setter defined for it, using that method is the easiest and most efficient way to set an attribute.

The following sample code demonstrates using an explicit setter method, setDateofbirth().

```
Date now = new Date();
def.setDateofbirth(now);
```

Using the Generic setAttribute() Method

Because specific setter methods are not available for all attributes, there is a generic setAttribute() method to fall back on.

To set the value of an attribute, use the LibraryObject.setAttribute() method. This method takes two arguments:

- The name of the required attribute. This argument should be supplied using the static variable defined in the instance Bean.
- An instance of oracle.ifs.common.AttributeValue that contains the new value for the attribute.

To create the AttributeValue object, use the static method AttributeValue.newAttributeValue(). This method is overloaded to allow it to handle each of the possible Oracle *i*FS datatypes.

Using setAttribute() is a two-step process:

1. Create an attribute object containing the new value of the attribute using the AttributeValue.newAttributeValue() method.

   The newAttributeValue() method is overloaded to handle each of the possible Oracle *i*FS datatypes.

2. Pass the attribute name and the attribute object to the setAttribute() method.

The following sample code demonstrates the two steps:

```
AttributeValue av = AttributeValue.newAttributeValue("blue");
setAttribute(FAVORITECOLOR_ATTRIBUTE,av);
```

As a matter of good coding practice, you can supply the attribute name using a static variable defined in the instance Bean, as demonstrated by `FAVORITECOLOR_ATTRIBUTE` in the example above.

Both setAttribute() and newAttributeValue() can throw oracle.ifs.common. IfsException.

Defining Explicit Getters and Setters

It is good programming practice to always provide getter and setter methods for each attribute defined by a custom type. These methods are simply wrappers for the generic getAttribute() and setAttribute() method. When defining explicit getter and setter methods in an instance Bean, access the current LibrarySession using the inherited method PublicObject.getSession().

Sample Code: Defining an Explicit Getter Method

```
public String getFullname()
  throws IfsException
{
   AttributeValue av = getAttribute(FULLNAME_ATTRIBUTE);
   return av.getString(getSession());
}

public Date getDateofbirth()
  throws IfsException
{
   AttributeValue av = getAttribute(DATEOFBIRTH_ATTRIBUTE);
   return (Date) av.getDate(getSession());
}
```

Sample Code: Defining an Explicit Setter Method

```
public void setFullname(String newValue)
  throws IfsException
{
   AttributeValue av = AttributeValue.newAttributeValue(newValue);
   setAttribute(FULLNAME_ATTRIBUTE,av);
}

public void setDateofbirth(Date newValue)
  throws IfsException
{
   AttributeValue av = AttributeValue.newAttributeValue(newValue);
   setAttribute(DATEOFBIRTH_ATTRIBUTE,av);
}
```

Searching for a Document

To perform a simple attribute-based search of the repository, use an instance of the class oracle.ifs.bean.Selector. The Selector class creates and executes simple searches. That is, Selectors search only one class (no joins are supported).

To perform a search:

1. Specify the search criteria.

   The search selection takes the form of a SQL WHERE clause (without the word "WHERE"). For example, to search based on the Name attribute, use the following criteria:

   ```
   NAME_ATTRIBUTE + "= '" + name + "'"
   ```

   To search for Owner or Creator, you need the DirectoryUserID:

   ```
   OWNER_ATTRIBUTE + "=" + directoryUser.getId();
   ```

2. Construct a Selector object, passing in the class to be searched and the search criteria.
3. Call the getItems() method to run the query.

Note that instances of this class do not persist, but must be created for each search.

A Selector object is created using the constructor defined in the Selector class. This constructor expects three arguments:

Name	Datatype	Description
`session`	LibrarySession	Current instance of LibrarySession.
`name`	String	Name of the class containing the attributes being searched on.
`search`	String	Search criteria.

The search is performed the first time a method is invoked that accesses the result set.

Specific Search Methods

To specify additional information about the search, use the following methods:

Method	Description
`Selector.setRecursiveSearch()`	Use to specify that the search should include objects of this class and also objects of its subclasses.
`Selector.setSearchSelection()`	Use to change the search criteria. This method expects to be passed a single argument that defines the new search criterion.
`Selector.setSortSpecification()`	Use to control the ordering of search results.

Sample Code: Attribute-based Search

```
public static void findMatchingFiles(LibrarySession ifsSession, String name)
   throws IfsException
{
   String search = PublicObject.NAME_ATTRIBUTE
                 + "= ’" + name +"’";
   Selector mySelector = new Selector (ifsSession, Document.CLASS_NAME, search);
   LibraryObject[] objs = mySelector.getItems();
   int count = (objs == null) ? 0 :objs.length;
   if (count == 0)
   {
     System.out.println("Search did not find any documents.");
   }
   else
   {
     for (int i = 0; i < count; i++)
     {
        Document myDocument = (Document)objs[i];
        String path = myDoc.getAnyFolderPath();
        System.out.println("Found a document at: " + path);
     }
   }
}
```

Sample Code: Hello World

The following code example demonstrates:

- Connecting to the repository.
- Constructing a new DocumentDefinition.
- Creating a new Document.
- Obtaining the current user’s home folder.
- Adding the new Document to the home folder.
- Disconnecting from the repository.

```
public static void HelloWorld()  throws IfsException
 {
   //Connect to the repository.
   LibraryService ifsService = new LibraryService();

   CleartextCredential me = new CleartextCredential(username, password);
   ConnectOptions connectOpts = new ConnectOptions();
   connectOpts.setServiceName(servicename);
   connectOpts.setServicePassword(servicepassword);

   LibrarySession ifsSession = ifsService.connect(me,connectOpts);

   //Create a new DocumentDefinition and a new Document.
   DocumentDefinition newDocDef = new DocumentDefinition(ifsSession);
   newDocDef.setAttribute("NAME", AttributeValue.newAttributeValue
      ("Hello_World.txt"));
   newDocDef.setContent("Hello World");
   Document doc = (Document) ifsSession.createPublicObject(newDocDef);

   //Obtain the user’s home folder and add the new Document to it.
   DirectoryUser thisUser = ifsSession.getDirectoryUser();

   PrimaryUserProfile userProfile = ifsSession.getPrimaryUserProfile(thisUser);
   Folder homeFolder = userProfile.getHomeFolder();

   homeFolder.addItem(doc);

   //Disconnect from the repository.
   ifsSession.disconnect();
}
```

4 Creating Custom Classes

This chapter covers the following topics:

Overview of Creating Custom Classes

The most frequent customization within Oracle *i*FS involves adding custom attributes to the existing Document class. If an application requires only such basic customization, you can use XML to define a custom type. Creating a type definition file in XML corresponds roughly to subclassing the Document class in Java. Because creating "custom classes" is a familiar concept, we have used that term as well as the less familiar "custom types." (Note this slight distinction: in XML, the Superclass reference is to "Document", while in Java, the instance Bean extends "TieDocument." For more information, see "Tie Classes" in Chapter 2, "API Overview".)

Creating a custom class is the most basic Oracle *i*FS customization. Other ways to customize Oracle *i*FS include writing custom parsers, renderers, JSPs, and agents.

In Oracle *i*FS, custom classes (custom types) work as follows:

1. The developer creates a custom class and loads it into Oracle *i*FS.

   The custom class must be loaded before any document instances can be created.
2. Users create and load instances of the newly defined class into Oracle *i*FS.
3. Oracle *i*FS automatically parses the instances of the custom class and stores their elements as attributes.

   Once instances are stored in the repository, end users can execute queries on the attributes.

Creating a Type Definition File

If you have a group of documents with custom attributes that you want Oracle *i*FS to recognize and manipulate, you need to be able to describe these documents to Oracle *i*FS. To describe a custom document to Oracle *i*FS, use XML to create a type definition file.

How Do Type Definitions Work?

A type definition is based on a simple inheritance model in which a new type (class) inherits the attributes and behavior associated with its superclass. The Oracle *i*FS class hierarchy is based on a single inheritance tree:

```
LibraryObject     //The root class for all of iFS.
    PublicObject   //The root class for all user-related classes.
        Document   //The subclass of PublicObject that includes content.
```

Because the Document class is used for all objects that include content, Document is the most frequently used superclass. For a complete listing of the Oracle *i*FS API class hierarchy, see the *Oracle Internet File System Setup and Administration Guide.*

You can think of a type definition as consisting of two logical parts:

- Description: The first section describes the new content type, specifying its name and superclass.
- Attributes: The second section lists the attributes of the new content type, including one entry for each attribute defined. Each entry specifies the attribute label and data type.

The Type Definition File: Description Section

The Description section of the type definition includes six tags:

- The <ClassObject> Tag
- The <Name> Tag
- The <Description> Tag
- The <Superclass> Tag
- The <BeanClassPath> Tag
- The <ServerClassPath> Tag

This code fragment consists of a Description section, followed by information about each tag:

```
<ClassObject>
  <Name>InsuranceForm</Name>
  <Description>Claim Form</Description>
  <Superclass Reftype ="name">Document</Superclass>
 .
 .
 .
 </ClassObject>
```

The <ClassObject> Tag

The first tag in a type definition file must be <ClassObject>. The <ClassObject> and </ClassObject> tags encapsulate the type definition and specify to Oracle *i*FS that you are creating a new repository object of the ClassObject class. The ClassObject class contains a registration entry for each class in the Oracle *i*FS API. For example, there are ClassObject objects defined for the Document class and the Folder class. The ClassObject is used internally to manage instances of the class. Thus, for each custom type definition, a new ClassObject entry is created.

The <Name> Tag

When you insert the type definition into the repository, Oracle *i*FS uses the data you provide to define a table in the repository to store information about documents of the custom type. The value of the <Name> tag is used to construct the table name, so it must not include spaces.

The <Description> Tag

Optional. The <Description> tag provides a multi-word description of the custom type.

The <Superclass> Tag

A type definition file creates a new class in the repository by subclassing an existing class, usually the Document class. Use the <Superclass> tag to specify the name of the superclass.

The <Superclass> tag requires a RefType parameter to specify that you are going to provide the name of the superclass, rather than referencing it in any other way, such as by using an object ID. Here is an example of the <Superclass> tag in use:

```
<Superclass RefType='name'>Document</Superclass>
```

The <BeanClassPath> Tag

Optional. The <BeanClassPath> tag specifies the fully qualified path to the instance class Bean. Required only if a custom instance Bean is written.

The <ServerClassPath> Tag

Optional. The <ServerClassPath> tag specifies the fully qualified path to the server class Bean, which by convention is an S_ class. Required only if a custom server-side override is written.

The Type Definition File: Attributes Section

The second section of the type definition is the Attributes section, enclosed by a pair of <Attributes> </Attributes> tags. The Attributes section consists of custom attribute definition entries. The Attributes section contains one element for each attribute that is unique to this type. Each attribute definition entry is enclosed by a pair of <Attribute> </Attribute> tags, and may include three tags:

- The Attribute <Name> Tag
- The <DataType> Tag
- The <DataLength> Tag
- The <ClassDomain> Tag

Here is an attribute definition entry for the ClaimNumber attribute:

```
<Attribute>
  <Name>ClaimNumber</Name>
  <DataType>Long</DataType>
</Attribute>
```

The Attribute <Name> Tag

The attribute <Name> tag specifies the name of the attribute being defined. This is the name of the variable that will be used in the getter and setter methods in the corresponding instance class Bean. Internally, Oracle *i*FS converts all attribute names to uppercase, so you can use any combination of case that you choose, but you cannot have two names that differentiate only by case. Thus, `FullName`, `FULLNAME`, `fullname`, and `fullName` are all legal, but having two attributes called `FullName` and `FULLNAME` is not.

The <DataType> Tag

The <DataType> tag specifies the type of data stored in the attribute. For example, "String" and "integer" are common datatypes.

The <DataLength> Tag

Optional, used for String values only. The <DataLength> tag specifies the maximum length in bytes for String attributes. Values greater than this number will result in an error message.

The <ClassDomain> Tag

The <ClassDomain> tag is used to restrict the types of objects that can be stored in an attribute whose datatype is PublicObject. For more information, see "Sample Code: ClassDomain Definition".

Sample Code: Create a Type Definition

The completed type definition looks like this:

```
<?xml version = '1.0' standalone = 'yes'?>
<!-- CreateInsuranceForm.xml -->
<ClassObject>
   <Name>InsuranceForm</Name>
   <Description>Claim Form</Description>
   <Superclass Reftype ="name">Document</Superclass>
   <Attributes>
      <Attribute>
          <Name>ClaimNumber</Name>
          <DataType>Long</DataType>
      </Attribute>

      <Attribute>
          <Name>ClaimType</Name>
          <DataType>String</DataType>
          <DataLength>50</DataLength>
      </Attribute>

      <Attribute>
          <Name>InsuranceFormStreetAddress</NAME>
          <DataType>PublicObject</DataType>
         <ClassDomain RefType="Name">InsuranceFormStreetAddressDomain
          </ClassDomain>
    </Attributes>
</ClassObject>
```

In this example, the attribute, InsuranceFormStreetAddress, is an embedded attribute defined as datatype PublicObject and restricted to containing objects of the class InsuranceFormStreetAddressDomain.

Using Compound Attributes

An attribute in an XML file can be:

- A simple value, such as a person's name.
- A compound value, such as an address, with its elements of Street, City, and State, which is derived from another object and embedded in the parent object as a component.

To handle simple values, use the database datatypes, such as String, Integer, Long, Date.

To handle compound values, follow these steps:

1. Create a separate type definition for each compound attribute. For example, you could create an Address object that defined attributes for Street, City, State, Zip, and Country. Typically, you would subclass the ApplicationObject class, which inherits from PublicObject. See "Sample Code: Embedded Attribute Type Definition".
2. Define a ClassDomain object based on the type that was just created. See "Sample Code: ClassDomain Definition".
3. In the parent type definition, define the compound attribute as datatype PublicObject and use the ClassDomain object to restrict the kind of PublicObjects this attribute can contain to instances of the InsuranceFormStreetAddress class. See "Sample Code: Create a Type Definition".

Sample Code: Embedded Attribute Type Definition

In this example, a new class object, InsuranceFormStreetAddress, defines the elements of a compound attribute that includes Street, City, State, Zip, and Country.

```
<?xml version = '1.0' standalone = 'yes'?>
<!-- InsuranceFormStreetAddress.xml -->
<ClassObject>
   <Name>InsuranceFormStreetAddress</Name>
   <Description>Insurance Form Address Definition</Description>
   <Superclass Reftype ="name">ApplicationObject</Superclass>

   <Attributes>
      <Attribute>
          <Name>AddressType</Name>
          <DataType>String</DataType>
          <DataLength>64</DataLength>
```

```
  </Attribute>

  <Attribute>
      <Name>StreetLine1</Name>
      <DataType>String</DataType>
      <DataLength>64</DataLength>
  </Attribute>

  <Attribute>
      <Name>StreetLine2</Name>
      <DataType>String</DataType>
      <DataLength>64</DataLength>
  </Attribute>

  <Attribute>
      <Name>StreetLine3</Name>
      <DataType>String</DataType>
      <DataLength>64</DataLength>
  </Attribute>

  <Attribute>
      <Name>City</Name>
      <DataType>String</DataType>
      <DataLength>64</DataLength>
  </Attribute>

  <Attribute>
      <Name>State</Name>
      <DataType>String</DataType>
      <DataLength>32</DataLength>
  </Attribute>

  <Attribute>
      <Name>Zip</Name>
      <DataType>String</DataType>
      <DataLength>16</DataLength>
  </Attribute>

  <Attribute>
      <Name>Country</Name>
      <DataType>String</DataType>
      <DataLength>32</DataLength>
  </Attribute>

</Attributes>
```

```
  </ClassObject>
```

Sample Code: ClassDomain Definition

This class domain restricts entries to be the ClassObject defined in InsuranceFormStreetAddress.xml.

```
<?xml version="1.0" standalone="yes"?>
<!--InsuranceFormStreetAddressDomain.xml-->
<ClassDomain>
  <Name>InsuranceFormStreetAddressDomain</Name>
  <DomainType>1</DomainType>
  <Classes>
    <ArrayElement reftype="name">InsuranceFormStreetAddress</ArrayElement>
  </Classes>
</ClassDomain>
```

Load a Custom Type Definition

To load a custom document type, log in as an administrator and upload the type definition file into Oracle *i*FS. You can load a type definition file to any appropriate folder in Oracle *i*FS; no special location is required.

Creating an Instance Class Bean

Once you've created a custom type definition, (although not required) that you create a corresponding instance class Bean to implement custom behavior and convenience methods for getting and setting custom attributes.

The instance class Bean is recommended for coding convenience, as it provides the expected getter and setter methods for the custom attributes. However, it is not required unless you need to provide unique functionality for this type.

The instance class Bean should include:

- A standard constructor, consisting of a call to "super" (the superclass, Document).
- Getter and setter methods for attributes defined in the type definition file.
- Any methods needed to provide functionality unique to the type.

Note the following connections between the type definition file and the instance class Bean:

- The instance class Bean extends the implementation class of the superclass specified in the type definition file. Tie classes provide implementation classes for all Oracle *i*FS classes. Thus, because the CreateInsuranceForm.xml type definition file defines Document as the superclass, the CreateInsuranceForm.java instance class Bean extends TieDocument, which is the implementation class for Document.
- For more information about using Tie classes, see "Tie Classes" in Chapter 2, "API Overview".
- The name and package of the instance class Bean must match the value of the <BeanClassPath> attribute in the type definition file, if one is present.

Sample Code: Create an Instance Class Bean

```
/*---CreateInsuranceForm.java---*/
package ifsdevkit.sampleapps.insurance;

import oracle.ifs.beans.ClassObject;
import oracle.ifs.beans.DirectoryUser;
import oracle.ifs.beans.FolderPathResolver;
import oracle.ifs.beans.LibrarySession;
import oracle.ifs.beans.PublicObject;
import oracle.ifs.common.AttributeValue;
import oracle.ifs.common.IfsException;

public class InsuranceBean
{
  /**
   * The name for the claim.
   */
  protected String m_name;

  /**
   * The type of the claim.
   */
  protected String m_claimType;

  /**
   * The claim number.
   */
  protected Long   m_claimNumber;

  /**
   * Constructor
```

```
    */
   public InsuranceBean()
   {
   }

   /**
    * Initialize the bean and populate the necessary fields.
    *
    * @param session   The <code>LibrarySession</code> object.
    *
    * @param resolver  The <code>FolderPathResolver</code> object.
    *
    * @param path  The path to the insurance object.
    *
    * @exception IfsException Thrown if operation failed.
    */
   public void init(LibrarySession session, FolderPathResolver resolver, String
path)
   throws IfsException
   {
     try
     {
       PublicObject insuranceObj = resolver.findPublicObjectByPath(path);
       ClassObject co = insuranceObj.getClassObject();

       m_name = insuranceObj.getName();
       AttributeValue av = insuranceObj.getAttribute("CLAIMTYPE");
       if (!av.isNullValue())
       {
         m_claimType = av.getString(session);
       }
       av = insuranceObj.getAttribute("CLAIMNUMBER");
       if (!av.isNullValue())
       {
         m_claimNumber = new Long(av.getLong(session));
       }
     }
     catch (IfsException e)
     {
       e.printStackTrace();
       throw e;
     }
   }

   /**
```

```
     * Return name for the claim.
     *
     * @return The claim name.
     */
    public String getName()
    {
      return m_name;
    }

    /**
     * Return the claim type.
     *
     * @return The claim type in a <code>String</code>.
     */
    public String getClaimType()
    {
      return m_claimType;
    }

    /**
     * Return the claim number.
     *
     * @return The claim number as a <code>Long</code>.
     */
    public Long getClaimNumber()
    {
      return m_claimNumber;
    }
  }
```

Deploy an Instance Class Bean

For the protocol servers and other standard Oracle *i*FS components to access your custom instance class Bean, the folder tree containing the class for the Bean must reside in the Oracle *i*FS CLASSPATH. Oracle *i*FS includes a special directory for this purpose. This directory, called `custom_classes`, is already in the CLASSPATH environment variable that the Oracle *i*FS server software uses.

To deploy an instance class Bean:

1. Compile the instance class Bean, creating a `.class` file.

2. Place the folder tree that contains the resulting `.class` file in the directory `$ORACLE_HOME/ifs/custom_classes` on the server where Oracle *i*FS is installed.

For example, if the compiled Bean, `InsuranceForm.class`, is stored in the package `ifs.examples`, the entire folder tree, from `/ifs` on, must be stored in the `custom_classes` directory.

Note: The compiled Java Bean must be copied to the native file system of the server, not to the Oracle *i*FS repository.

Creating Document Instances

Once you have created the custom type file to define the custom class for the insurance form document and stored it in Oracle *i*FS, you can create document instance files to instantiate the class. These document instances can be either in a custom format or in XML.

When you upload document instance files into Oracle *i*FS, Oracle *i*FS parses the files and creates the appropriate objects, which are stored in the Oracle *i*FS repository. The specific parser called varies according to the way the files are loaded, as shown in the following table. For more information about using and registering parsers, see Chapter 5, "Using Parsers".

Document Instance Format	Loaded By	Parsed By
XML	Any Oracle *i*FS protocol or user interface	Automatically parsed by the SimpleXmlParser.
XML	Application program	SimpleXmlParser must be explicitly called by the application.
Custom	Any means: protocol, user interface, or application program	Custom parser must be written and registered for use by protocols and user interfaces, or explicitly called by the application.

Note: Once Oracle *i*FS creates objects from the document instance files, the document instance files themselves are not stored in the Oracle *i*FS repository. If you think you may need further access to the document instance files (for example, to modify them to create new document instances), you may want to keep copies of the instance files on your local drive.

Sample Code: Create Document Instances

These two files, `claim1.xml` and `claim2.xml`, are the document files that create two specific instances of the InsuranceForm class.

```
<?xml version = '1.0' standalone = 'yes'?>
<!-- claim1.xml -->
<InsuranceForm>
   <Name>Juana Angeles</Name>
   <ClaimNumber>35093</ClaimNumber>
   <ClaimType>Car Accident</ClaimType>
   <FolderPath>/public/examples/insuranceApp/claims</FolderPath>
</InsuranceForm>

<?xml version = '1.0' standalone = 'yes'?>
<!-- claim2.xml -->
<InsuranceForm>
   <Name>Kevin Chu</Name>
   <ClaimNumber>41111</ClaimNumber>
   <ClaimType>Car Accident</ClaimType>
   <FolderPath>/public/examples/insuranceApp/claims</FolderPath>
</InsuranceForm>
```

Upload Document Instance Files

Use one of these options to upload document instance files:

- Use a standalone FTP product.
- In the Oracle *i*FS Web interface, use either:
  - `Upload via Drag and Drop` or
  - `Upload via Browse` with `Parse on Upload` selected.

Limitations on XML Type Definition Files

If you are creating XML type definition files, you should be aware of the following limitations:

- Although you can use XML to *create* custom type files, you cannot use XML to *delete* or *modify* custom type files.
- Use Oracle *i*FS Manager to delete or modify custom type files.

For information about using Oracle *i*FS Manager, see the *Internet File System Setup and Administration Guide.*

5 Using Parsers

This chapter covers the following topics:

What Is a Parser?

A parser is a Java class that extracts attributes from a local file and stores the information in the repository. More specifically, in the case of a document, a parser:

- Takes in an InputStream or Reader object.
- Processes the character input, extracting attributes as it goes.
- Produces one or more *i*FS objects, such as a Document or a Folder.

Note: Significant improvements have been made to the XML parsing capabilities of Oracle *i*FS for version 1.1. For more information, see What's New in Oracle *i*FS 1.1.

Standard Oracle *i*FS Parsers vs. Custom Parsers

Whether your application requires a custom parser depends upon the format of the documents produced by the application:

Document Format Produced	Parser Options
Standard XML documents	Use the Oracle *i*FS standard XML parser out-of-the-box.
Custom format	Write a custom parser.

Whether or not you must explicitly invoke a parser depends on how the documents produced by your application are entered into the Oracle Internet File System:

- If the documents are uploaded using the protocol servers, the Oracle *i*FS XML parser will be invoked automatically by the protocol servers.
- If the documents are uploaded by an application, you must explicitly invoke a parser, either the Oracle *i*FS XML parser or a custom parser. Otherwise, the documents will be read in as raw data, rather than parsed into objects.

If your application defines a custom class that produces documents in a special format that is not XML, you will need to create a custom parser using the classes and methods provided as part of the Oracle *i*FS Java API. This custom parser will create Oracle *i*FS repository objects of your custom class. For example, assume you have defined a Memo class that subclasses the Document class. The Memo class includes the following custom attributes: To, From, Date, and Text (the content of the memo). To store Memo objects in Oracle *i*FS requires a parser. If the Memo documents are in XML, you can use the Oracle *i*FS SimpleXmlParser to extract the attributes. If the Memo documents are stored in a special format, you will need to create a custom parser and specify how it is to extract the attributes.

Using the Standard Parsers

Out-of-the-box, Oracle *i*FS includes several standard parsers that will meet most needs of developers creating new applications in Oracle *i*FS.

The following table lists the Oracle *i*FS standard parser classes.

Class	Description
SimpleXmlParser	Creates an object in the Oracle *i*FS repository from an XML document body. Used as the default parser for all XML documents stored in Oracle *i*FS. SimpleXmlParser extends XmlParser.
XmlParser	A base class for custom XML parser development.
ClassSelectionParser	Adds custom attributes to all files of a specified format. Performs no actual parsing.

Parsing Options

To understand the parsing options provided by Oracle *i*FS, consider XML files in 3 categories:

XML files meant to configure Oracle iFS

XML files to be parsed

XML files to be stored without parsing

Using the SimpleXmlParser

If your customization requirements are minimal, you can define a custom class using XML to add custom attributes. Once you create the type definition file, including any custom attributes, the SimpleXmlParser automatically recognizes the custom attributes and parses them correctly.

When custom XML documents are added to Oracle *i*FS using any of the protocols or user interfaces, those documents are automatically parsed by the SimpleXmlParser, without any further custom coding.

Specifically, the SimpleXmlParser works as described above for FTP, SMB, the Windows interface, and the Oracle *i*FS Web interface using `Upload via Drag` `and Drop`. If you prefer to use the Oracle *i*FS Web interface `Upload via Browse` facility, you need to also click the checkbox for `Parse File on Upload`.

If your XML documents are added to Oracle *i*FS by an application, the application must explicitly invoke the SimpleXmlParser.

The ClassSelectionParser

The ClassSelectionParser is unique in that it does not perform any actual parsing. Rather, the ClassSelectionParser allows you to add one or more custom attributes to files with a specific file extension, such as all `.doc` files, before the files are stored in the repository. The ClassSelectionParser provides the mechanism for mapping a class to a specific file format. For more information, see "Using the ClassSelectionParser".

Using the ClassSelectionParser

Implementing a ClassSelectionParser is a three-step process:

1. Create a Class Definition
2. Register the Extension with the ClassSelectionParser
3. Register the Class

Create a Class Definition

The first step in implementing a ClassSelectionParser is to create the custom class definition. This example defines a custom class for presentation slides, and describes one additional attribute, "NumberOfSlides," to be added to all files with the file extension `.ppt`.

```
<?xml version = '1.0' standalone = 'yes'?>
<!--Presentation.xml-->
<ClassObject>
  <Name>Presentation</Name>
   <Superclass RefType='Name'>Document</Superclass>
  <Description>Custom Class for Presentations</Description>
  <Attributes>
    <Attribute>
      <Name>NumberOfSlides</Name>
      <DataType>INTEGER</DataType>
    </Attribute>
  </Attributes>
</ClassObject>
```

Register the Extension with the ClassSelectionParser

Once the custom class has been created, associate the file extension (`ppt`) with the parser (`ClassSelectionParser`) by the usual registration process. You can register the parser using Oracle *i*FS Manager or XML.

```
<?xml version = '1.0' standalone = 'yes'?>
<!--RegisterPPTParser.xml-->
<PropertyBundle>
  <Update Reftype='ValueDefault'>ParserLookupByFileExtension</Update>
  <Properties>
    <Property Action = 'add'>
      <Name>ppt</Name>
      <Value Datatype='String'>
          oracle.ifs.beans.parsers.ClassSelectionParser
      </Value>
    </Property>
  </Properties>
</PropertyBundle>
```

Register the Class

Once the parser has been registered, you must register the custom class by adding an entry to the IFS.PARSER.ObjectTypeLookupByFileExtension PropertyBundle. Just as registering a parser requires adding an entry to a PropertyBundle, so registering a class also requires adding an entry to a PropertyBundle. In this case, the registration process associates the file extension (.`ppt`) with the custom class (`Presentation`). You only need to specify the actual class name, not the fully qualified path name.

Registering the class completes the process necessary to invoke the ClassSelectionParser. If this step is omitted, the class associated with the ClassSelectionParser defaults to Document; no parsing will occur.

```
<?xml version = '1.0' standalone = 'yes'?>
<!--RegisterPPTObjectType.xml-->
<PropertyBundle>
  <Update Reftype='ValueDefault'>IFS.PARSER.ObjectTypeLookupByFileExtension
</Update>
  <Properties>
    <Property Action = 'Add'>
      <Name>ppt</Name>
      <Value Datatype='String'>Presentation</Value>
    </Property>
  </Properties>
</PropertyBundle>
```

How Does XML Parsing Work?

When you place an XML representation of a document in Oracle *i*FS, the SimpleXmlParser is called to create the document object. The following table provides an overview of how parsing an XML document works.

Step	Who	Does What	Result
1.	User	Loads a local file using any *i*FS interface or supported protocol.	MyDocument.xml is loaded into *i*FS.
2.	Interface or Protocol	Performs parser lookup based on the file extension.	If there is no corresponding parser, the document is simply stored "as is," with the content and attributes from Step 1. Or, if there is a parser defined for the file extension, that parser is invoked.
3.	SimpleXmlParser	Parses the XML file.	Creates an object of the type defined in the XML file's <ClassName> tag: - A new class, if the value is `ClassObject`. - An instance of an Oracle *i*FS standard class, such as `Document`. - An instance of a custom class.

To illustrate this sequence, consider the following example.

1. An end user drags a document instance, such as `claim3.xml`, into an Oracle *i*FS folder, `/ifs/system/claims`.
2. SMB performs a parser lookup based on the file extension, `.xml`. Because this is an XML file, the parser lookup finds a match and invokes the SimpleXmlParser.
3. Because the claim custom class definition file was previously stored in Oracle *i*FS, and because the XML file's Root Element has the same name as the name of the claim custom type, the SimpleXmlParser recognizes `claim2.xml` as an instance of `claim.xml`. The SimpleXmlParser parses `claim2.xml`, creating an object called `claim2`.

Using a Custom Parser

If you want to parse non-XML documents, such as `.doc` or `.xls` documents, you must write a custom parser to create database objects from these documents. To create a custom parser, you can either subclass an existing Oracle *i*FS parser or create a custom class from scratch, implementing the `oracle.ifs.beans.parsers.Parser` interface.

The Parser class creates one or more objects. In most cases, the Parser class is used to create the following objects:

- Documents
- Folders
- A combination of documents and folders

A parser determines which type of object to create based on the InputStream or Reader object passed to it. If the InputStream or Reader describes more than one type of object, the parser can either:

- Create each object as soon as it is complete.
- Create all objects upon reaching the end of the stream.

Overview of a Parser Application

A parser application includes four components:

- The application that calls the parser (the "parser application").
- The parser itself.
- The ParserCallback (optional).
- The mechanism for registering a parser, the ParserLookupByFileExtension PropertyBundle.

The following table describes each component.

Component	Description/Sample
Application	The application creates an instance of the parser required, then calls the parser, specifying the document representation (required), the name of the ParserCallback object (optional), and the Options object (optional).
Parser	The parser executes whatever custom code is needed to create the parsed object, then stores the parsed object in the repository.
ParserCallback	The application may optionally specify a ParserCallback object. The ParserCallback object's preOperation() or postOperation() methods specify additional processing that is executed before, after, or both before and after the parsing operation takes place.
ParserLookupBy FileExtension PropertyBundle	Oracle *i*FS looks up the name of the parser for this document class in the ParserLookupByFileExtension PropertyBundle.

Writing a Parser Application

Writing a parser application include the following stages:

1. Write the Parser Class
2. Deploy the Parser
3. Invoke the Parser (in the parser application)
4. Write a ParserCallback (optional)

For a short parser example, see "Sample Code: A Custom Parser", in this chapter.

For more information about parsers, see the following classes in the Oracle *i*FS Javadoc:

- `oracle.ifs.beans.parsers.SimpleXmlParser`
- `oracle.ifs.beans.parsers.XmlParser`
- `oracle.ifs.beans.parsers.SimpleTextParser`

Write the Parser Class

The purpose of a parser is to identify the properties in a file, and use the properties to create a database object.

When creating a custom parser, you can choose from two approaches:

- If one of the existing Oracle *i*FS parsers partially meets the needs of your application, you can subclass an existing parser. (SimpleXmlParser extends

XmlParser, which implements the `oracle.ifs.beans.parsers.Parser` interface.)

- If you cannot use an existing Oracle *i*FS parser as a starting point, you must create a custom class from scratch, directly implementing the interface `oracle.ifs.beans.parsers.Parser`.

Whichever approach you choose, writing a custom parser means implementing the Parser interface, either directly or indirectly. The Parser interface includes one overloaded method, parse(), which accepts two types of input:

- An InputStream object
- A Reader object

Once the parse() method has been called, the balance of the code of the parser itself examines each line and places its content into the appropriate attribute of the object the parser is creating. The syntax and arguments for parse() are described below.

To write a custom parser, you must write two methods:

- A constructor
- A parse() method

Write a Constructor

Every parser must implement the standard constructor for a parser. The standard constructor takes one parameter, as shown in the following table.

Parameter	Datatype	Description
`session`	LibrarySession	The LibrarySession of the current user.

Sample Code: A Constructor

```
public SimplestParser(LibrarySession lib) throws IfsException
```

Write a parse() Method

The following table describes the parameters of the parse() method.

Parameter	Datatype	Description
`stream`	InputStream	An InputStream for the parser to read. Use an InputStream for data that is not character-based, such as audio and video data.
`reader`	Reader	Alternatively, a Reader for the parser to read. A Reader should be used for character-based data.
`callback`	ParserCallback	Optional parameter. May be null. If specified, the ParserCallback object includes methods that specify processing to be implemented before parsing, after parsing, or both.
`options`	Hashtable	Optional parameter. May be null. If specified, the Options parameter further controls the behavior of the parser through a set of optional name/value pairs. Commonly used to specify character encoding.

For sample code for the parse() method, see "Sample Code: A Custom Parser" in this chapter.

Overview of a Custom Parser

For a custom parser, see "Sample Code: A Custom Parser". This SimplestParser extracts the text between the <TITLE> tags of an HTML document and stores that information in a custom field. This requires that a subclass of Document, named CUSTOM with the the attribute TITLE, be registered on the server with the file extension .cus.   Customizing the file extension is for illustration purposes, only; you could register the file extension as .htm to use a similar parser for real HTML documents.

This is a simplified example and does not take into consideration versioned documents, nor does it address issues concerning local character sets.

Sample Code: A Custom Parser

```
package simparser;

// These classes provide the building blocks for an iFS document.

import oracle.ifs.beans.Attribute;
import oracle.ifs.beans.Document;
import oracle.ifs.beans.DocumentDefinition;
import oracle.ifs.beans.Format;
import oracle.ifs.beans.LibraryObject;
import oracle.ifs.common.Collection;
import oracle.ifs.common.AttributeValue;

// These classes are used to instantiate a folder object to store the document.

import oracle.ifs.beans.Folder;
import oracle.ifs.beans.FolderPathResolver;

// These classes are used to obtain information about the user at runtime.

import oracle.ifs.beans.DirectoryUser;
import oracle.ifs.beans.PrimaryUserProfile;
import oracle.ifs.beans.LibrarySession;

// These classes are the base classes for creating a parser.

import oracle.ifs.beans.parsers.Parser;
import oracle.ifs.beans.parsers.ParserCallback;
import java.util.Hashtable;

// These are standard Java objects used to process the document content.

import java.io.InputStream;
import java.io.InputStreamReader;
import java.io.Reader;
import java.io.BufferedReader;

// This class is used to report exceptions to iFS methods.

import oracle.ifs.common.IfsException;

public class SimplestParser implements Parser
{
  private String title;
  private LibrarySession m_librarySession;
  private Document newDoc;
  private Folder currentFolder;
  private Folder homeFolder;

// The constructor argument captures the current library session, which is
// used to pass information about the user and environment at runtime.
```

```
  public SimplestParser(LibrarySession lib) throws IfsException
  {
    m_librarySession = lib;
  }

/* This parser is called by the host protocol at runtime, passing a Reader
 * object with the contents of the document being parsed. The callback is
 * an optional argument that enables the parser to respond to the calling
 * method. The Hashtable is used to store three key parameters:
 * CURRENT_PATH_OPTION: the current working directory.
 * CURRENT_NAME_OPTION: the name of the file being parsed.
 * UPDATE_OBJECT_OPTION: indicates if the document being parsed is
 *                       replacing an object that already exists.
 */
  public LibraryObject parse(Reader htmlStream, ParserCallback callback,
        Hashtable options) throws IfsException
  {
  try
  {

/*  Instantiate a FolderPathResolver, then pass it the CURRENT_PATH_OPTION as
 *  a string. Set the currentFolder variable to the path where the document
 *  to be parsed was inserted.
 */

    FolderPathResolver fpr = new FolderPathResolver(m_librarySession);
    fpr.setRelativePath(options.get(CURRENT_PATH_OPTION).toString());
    currentFolder = fpr.getCurrentDirectory();

/*  Instantiate the string variable documentContent.
 *  Instantiate a BufferedReader object named dataStream and populate it
 *  with the document content passed in to the method.
 */

    String documentContent = "";
    BufferedReader dataStream =
                       new BufferedReader(htmlStream);

// Read the buffered data into the documentContent variable one line at a time.

    for (String line = dataStream.readLine();line != null;
                       line = dataStream.readLine())
    {
      documentContent = documentContent + line + "\n";
    }
```

```
//  Send the resulting string to the parseTitle method to extract the title.

    String docTitle = parseTitle(documentContent);

//  Instantiate a DocumentDefinition object.

    DocumentDefinition docDef = new DocumentDefinition(m_librarySession);

//  Instantiate a Collection object and populate it with the list of
//  format extensions. Set the format in the document definition.

    Collection allFormats = m_librarySession.getFormatExtensionCollection();
    docDef.setFormat((Format) allFormats.getItems("cus"));

//  The Classname is the name of the subclass we've defined (CUSTOM).

    docDef.setClassname("Customer");

//  Set the Name attribute in the document definition to the variable passed
//  to the parser in the options Hashtable.

    docDef.setAttribute("NAME", AttributeValue.newAttributeValue
                    (options.get(CURRENT_NAME_OPTION)));

//  Set the custom attribute "TITLE" to the docTitle variable returned by the
//  parseTitle method.

    docDef.setAttribute("TITLE", AttributeValue.newAttributeValue(docTitle));

//  Set the content of the document to the String documentContent.

    docDef.setContent(documentContent);

//  Instantiate a new Document using the DocumentDefinition just defined.

    Document newDoc = (Document) m_librarySession.createPublicObject(docDef);

/*  Check to see if the UPDATE_OBJECT_OPTION variable is set. If so, update
 *  the document (update). If not, create a new document (addItem).
*/
    if(options.get(UPDATE_OBJECT_OPTION) != null)
    {
      Document currentDoc = (Document) currentFolder.findPublicObjectByPath
                  (docDef.getAttribute("NAME").toString());
```

```
        currentDoc.update(docDef);
      }
      else
      {
        currentFolder.addItem(newDoc);
      }
    }

// Catch any exceptions. Set VerboseMessage to true to get a more complete
// report of the methods that threw the exception.

    catch (IfsException ifsExceptionCaught)
    {
      ifsExceptionCaught.setVerboseMessage(true);
      ifsExceptionCaught.printStackTrace();
    }
    catch (Exception exceptionCaught)
    {
      exceptionCaught.printStackTrace();
    }
    return newDoc;
    }

/* parse method called when the protocol sends the file content as an
 * InputStream. This method converts the InputStream to a BufferedReader
 * and forwards it to the first parse method (keeps code concise).
*/

    public LibraryObject parse(InputStream htmlStream, ParserCallback callback,
                               Hashtable options)
    {

// Convert the InputStream htmlStream to the BufferedReader named redirect.

      BufferedReader redirect =
          new BufferedReader(new InputStreamReader(htmlStream));

// Send the resulting BufferedReader to the first parse method.

      try {
        Document newDoc = (Document) parse(redirect,callback,options);
      }

// Catch and report (in verbose mode) any exceptions.
```

```
    catch (IfsException ifsExceptionCaught)
    {
      ifsExceptionCaught.setVerboseMessage(true);
      ifsExceptionCaught.printStackTrace();
    }
    catch (Exception exceptionCaught)
    {
      exceptionCaught.printStackTrace();
    }
    return newDoc;
  }

/*  This is the actual custom parsing routine. It searches the text String
 *  for the tag <TITLE>, starts at the 7th character (the length of the
 *  <TITLE> tag and extracts a substring of all the information through
 *  the last character before the </TITLE> tag.
 */

  private String parseTitle (String parseString){
    try
    {
      title = parseString.substring((parseString.indexOf("<TITLE>")+ 7),
                  parseString.indexOf("</TITLE>"));
    }
    catch (Exception e)
    {
      title = "Untitled";
      e.printStackTrace();
    }
    return title;
  }
}
```

Deploy the Parser

For the protocol servers and other standard Oracle *i*FS components to access your custom parser, the folder tree containing the class for the parser must reside in the Oracle *i*FS CLASSPATH. Oracle *i*FS includes a special directory for this purpose. This directory, called `custom_classes`, is already in the CLASSPATH environment variable that the Oracle *i*FS server software uses.

To deploy a parser:

1. Compile the parser, creating a `.class` file.
2. Place the folder tree that contains the resulting `.class` file in the directory `$ORACLE_HOME/ifs/custom_classes` on the server where Oracle *i*FS is installed.

Note: The compiled Java code must be copied to the native file system of the server, not to the Oracle *i*FS repository.

Register the Parser

The purpose of registering a parser is to map a certain file extension to a specific parser. Once this mapping is created, whenever a file with that extension is imported by an Oracle *i*FS client or protocol, the file will be passed to the custom parser before it is stored in the repository. You can register a parser in either of two ways:

Facility	Advantages/Restrictions
Oracle *i*FS Manager	Use Oracle *i*FS Manager for simplicity and ease-of-use. Using Oracle *i*FS Manager, you can only register a parser that exists on the same instance of Oracle *i*FS as the Oracle *i*FS Manager facility.
XML	Use XML if you prefer to register a parser using a script, or if you need to deploy the parser on a separate Oracle *i*FS instance.

Each registered parser has two attributes:

Attribute	Datatype	Description	Example
`Extension`	String	File extension.	`cus`
`ClassName`	String	Fully qualified classname of the parser.	`ifs.demo.SimplestParser .parser. SimplestParser`

The underlying mechanism for storing the mappings between file extensions and parsers is a PropertyBundle object called "ParserLookupByFileExtension." A PropertyBundle is a list of name/value pairs stored as an array of Property objects. Each Property object stores the mapping between a file extension and a parser as a Name/Value pair:

- The Name attribute of the Property stores the Extension attribute, such as `cus`.
- The Value attribute of the Property stores the ClassName of the parser, such as `ifs.demo.SimplestParser.parser.SimplestParser`.

Registering a Parser Using Oracle *i*FS Manager

To register a parser using Oracle *i*FS Manager, follow these steps:

1. From the Oracle *i*FS Manager Object menu, choose Register.
2. From the Select Object Type window, choose Parser Lookup.
3. From the Parser Lookup Registry window, choose Add.
4. In the Parser Lookup Entry window, fill in the text boxes for the attributes.
5. Click OK.

Registering a Parser Using XML

To register a parser using XML, write an XML file to add a new Property object to the ParserLookupByFileExtension PropertyBundle, specifying the file extension and class name of the parser.

```
<?xml version="1.0" standalone="yes"?>
<!--SimplestParser.xml-->
<PROPERTYBUNDLE>
   <UPDATE RefType="valuedefault">ParserLookupByFileExtension</UPDATE>
   <PROPERTIES>
      <PROPERTY ACTION="add">
         <NAME>po</NAME>
         <VALUE
DataType="String">ifs.demo.SimplestParser.parser.SimplestParser</VALUE>
      </PROPERTY>
   </PROPERTIES>
</PROPERTYBUNDLE>
```

Invoke the Parser

When an application program inserts content into the repository, the application is responsible for invoking the appropriate parser, either a standard Oracle *i*FS parser or a custom parser. (The protocols automatically call an Oracle *i*FS parser when they are used to insert documents into the repository.)

In order to parse a document, an application must:

- Instantiate the appropriate parser.
- Invoke the parse() method.

Write a ParserCallback

When a custom application calls a parser, the application may, optionally, pass in a ParserCallback object. A ParserCallback allows an application to provide additional processing before or after the actual parsing. The parser must, therefore, check to see if this optional parameter has been passed in.

The ParserCallback interface specifies three methods that allow an application to interact with a parser:

- The preOperation() Method
- The postOperation() method
- The signalException() method

The preOperation() Method

The application can use preOperation() to alter the LibraryObjectDefinition before the parser uses it to update the repository, in the following ways:

- To use the existing LibraryObjectDefinition, return the Definition as is.
- To change the object that will be parsed, construct a different Definition or modify the Definition before returning it.
- To prevent the parser from updating the repository, return a Definition value of "null".

Parameter Name	Datatype	Description
`lo`	LibraryObject	The object that will be updated by the parse operation. Value is "null" if the operation will create a new object.
`def`	LibraryObjectDefinition	The LibraryObjectDefinition that will be used to update the object, `lo`.

Sample Code: preOperation()

```
public LibraryObjectDefinition preOperation (LibraryObject lo,
                                             LibraryObjectDefinition def)
             throws IfsException
```

The postOperation() method

The application can use postOperation() to access the repository object that was created or updated by the parser, or to perform operations after the creation of a LibraryObject.

Parameter Name	Datatype	Description
`lo`	LibraryObject	The LibraryObject that was created or updated by the parse operation.

Sample Code: postOperation()

```
public void postOperation (LibraryObject lo)
             throws IfsException
```

The signalException() method

The application can implement signalException() to intercept any exceptions that occur during parsing. The options are:

- Throw the exception, in which case, parsing ceases.
- Simply return, in which case, parsing continues.

Parameter Name	Datatype	Description
e	IfsException	The potential exception.

Sample Code: signalException()

```
public void signalException(IfsException e)
             throws IfsException
```

Sample Code: ParserCallback Implementation

The following sample code provides a brief example of implementing the ParserCallback interface:

- Using the preOperation() method to create a folder before the object is parsed.
- Using the postOperation() method to add the parsed object to the folder after the object is parsed.

```
/*---FolderParsedObject.java---*/
private static class FolderParsedObject implements ParserCallback
{
    private Folder m_TargetFolder;
    public FolderParsedObject(Folder f)
    {
        m_TargetFolder = f;
    }
    public LibraryObjectDefinition preOperation(LibraryObject parm1,
                                              LibraryObjectDefinition parm2)
        throws IfsException
    {
        return parm2;
    }
    public void postOperation(LibraryObject newObject)
              throws IfsException
    {
        m_TargetFolder.addItem((PublicObject) newObject);
    }
    public void signalException(IfsException e)
        throws IfsException
    {
        throw e;
    }
}
```

6

Using Renderers

This chapter covers the following topics:

What Is a Renderer?

A renderer takes an object stored in the repository and outputs its content in a specific format. In a sense, a renderer is the opposite of a parser. While the information output by a renderer may be identical to the document as it was input, it doesn't have to be. You can use a renderer to:

- Reconstruct a parsed file and display it in its original format.
- Reconstruct a parsed file and display it in a different file format.
- Filter only certain file components for display.
- Calculate values from file components.

Once information in the original document has been parsed and stored as an object in the Oracle Internet File System, the object can be rendered in a variety of formats and layouts.

The Oracle *i*FS Framework for Rendering

The Oracle *i*FS rendering framework allows a developer to:

- Create a custom renderer by writing a class that implements the renderer interface.
- Register a custom renderer with the repository using XML files.
- Invoke a renderer using the methods inherited from LibraryObject. These methods can be used to invoke both out-of-the-box and custom renderers.

As a developer, you have two options to render a repository object:

- You can use the standard renderers provided by Oracle *i*FS. For more information, see "Using Standard Renderers".
- If the standard renderers do not meet the needs of your application, you can write a custom renderer in Java. For more information, see "Overview of a Renderer Application".

A Renderer Does Not Create a Repository Object

The output from a renderer is read-only and is not persistent. A renderer does *not* automatically create a Document object in the repository or a data file stored locally. If an application requires that the rendered output be available for later use, it is a post-rendering step to create a Document object or to save a data file locally.

What Objects Can Be Rendered?

Any LibraryObject can be rendered. For example, you might write an application that calls an XML renderer to display the following:

- If the object is a folder or an ACL, or other LibraryObject that does not contain content, the rendered output consists of attributes only.
- If the object contains content (that is, the object is a Document object or a subclass of Document), the rendered output could consist of attributes only, content only, or both.

Although, in general, any LibraryObject can be rendered, a custom renderer is likely to be written for the purpose of rendering specific kinds of objects. For example, while the SimpleXmlRenderer can render any LibraryObject, a PurchaseOrder renderer can render only PurchaseOrder objects, and throws an exception if requested to render an object that it cannot handle.

Because Document objects are the most commonly rendered objects, we will refer to documents for the balance of this discussion.

Using Server-Side Classes with Renderers

In the Oracle *i*FS Java class hierarchy each Oracle *i*FS object has two representations:

- The bean-side representation, known by the object name, such as Document.
- The server-side representation, known by the object name preceded with "S_", such as S_Document.

Because rendering is a server-side operation, you must use the S_ classes.

Using PolicyPropertyBundles to Register Renderers

The process of registering a renderer is, at the basic level, a matter of mapping a connection between a class of objects and a specific renderer. The underlying mechanism for storing these mappings is a PolicyPropertyBundle object.

A PolicyPropertyBundle is a specific type of PropertyBundle object. In general, PropertyBundles are used to store name/value pairs. In the case of a *`Policy`*PropertyBundle, a Policy is stored in each Property of the PropertyBundle. Each Policy contains the mapping between a class and a renderer for a specific protocol.

Each class in Oracle *i*FS has an associated PolicyPropertyBundle. When an object of a class is retrieved from the repository, Oracle *i*FS checks the associated

PolicyPropertyBundle to determine which renderer to use to display the object, based on the protocol making the request for the object.

Because Oracle *i*FS includes multiple protocols (FTP, HTTP, SMB), a specific Policy must be registered for each protocol; that is, one Policy each for HTTP, FTP, and SMB.

When a Property is used to store a Policy:

- The Name attribute of the Property object is specified first. The *Property* Name attribute must be the value of the *Policy* Operation attribute.
- The Value attribute of the Property object comes next. The *Property* Value attribute holds the Object ID of the *Policy* object.

Each Policy object stores the following attributes:

Attribute	**Datatype**	**Description**	**Example**
`Name`	String	Name of this custom renderer. Must be unique. Must be one word, no spaces.	`POSmbRenderer`
`Operation`	String	The name of the Property object. Operation specifies the key to the PolicyBundle hashtable, so it must precisely match the name used in the hashtable.	`SmbRenderer`
`ImplementationName`	String	Fully-qualified classname of the custom renderer, starting with package name.	`ifs.demo.po.renderer.PoRenderer`

For the procedures to register a renderer, see "Registering a Renderer Using Oracle iFS Manager". For an XML code sample of a PolicyPropertyBundle, see "Registering a Renderer Using XML".

Using Standard Renderers

Out-of-the-box, Oracle *i*FS includes two standard renderers that will meet the needs of most developers creating new applications in Oracle *i*FS.

Class	**Description**
SimpleXMLRenderer	Generates a complete XML document body based on properties in a particular document.
SimpleTextRenderer	Provided as a starting point example for developers who need to create a custom renderer.

Note: A renderer must be registered before you can use it. The standard renderers are registered by default.

Invoking Renderers

Whether your application invokes an Oracle *i*FS standard renderer or a custom renderer, the process is the same. Applications invoke renderers using the appropriate renderAs*Xxxx*() method defined in oracle.ifs.beans.LibraryObject. The renderer application must invoke the appropriate method for the type of output desired.

- An InputStream object (a series of bytes)
- A Reader object (a series of characters)

To use any renderer, invoke one of the following methods inherited from the LibraryObject class on the object you want to render:

```
public java.io.InputStream renderAsStream
                     (String rendererType,
                      String rendererName,
                      Hashtable options)
    throws IfsException

public java.io.Reader renderAsReader
                     (String rendererType,
                      String rendererName,
                      Hashtable options)
    throws IfsException
```

The following table lists the parameters for the renderAs*Xxxx*() methods. These parameters are used to determine which renderer is invoked and to pass options to the target renderer.

Parameter	Datatype	Description	Example
`rendererType`	String	Value of the Operation attribute of the Policy. Non-unique.	`SmbRenderer`
`rendererName`	String	Name of the Policy object. Must be unique.	`VcardSmbRenderer`
`options`	Hashtable	Contains values for each specific renderer to use.	Renderer-specific options. Values in this Hashtable must be serializable.

The `rendererType` and `rendererName` arguments differ as follows:

- The `rendererType` argument is a String value that specifies the name of a Policy's Operation attribute.
- The `rendererName` argument is a String value that specifies the Name attribute of the Policy that contains the Operation specified by `rendererType`.

The `rendererType` and `rendererName` arguments together determine which renderer is to be used. The determination is made as follows:

- If `rendererName` is not null:
  - The *custom* Policy object called `rendererName` for the Operation specified by `rendererType` is obtained. This custom Policy object contains the fully-qualified classname of the renderer in its ImplementationName attribute.
- If `rendererName` is null:
  - The *default* Policy object for this LibraryObject for the operation specified by `rendererType` is obtained. This default Policy object contains the fully-qualified classname of the renderer in its ImplementationName attribute.

Choosing a Renderer

You can use the `renderAs()` methods in two ways:

- Explicit specification: The application explicitly specifies a certain renderer.
- Default selection: The repository chooses a default renderer.

The following table shows the method signatures that correspond to each use:
‘

Use	Method Signatures

Explicit	`renderAsStream (String rendererType,String rendererName,`
	` Hashtable options)`
	`renderAsReader (String rendererType,String rendererName,`
	` Hashtable options)`

Default	`renderAsStream (String rendererType, null,`
	` Hashtable options)`
	`renderAsReader (String rendererType, null,`
	` Hashtable options)`

Example: Explicit Choice of Renderer

To choose a specific renderer, specify the `rendererName` attribute. The following example shows how an application can explicitly specify the SimpleTextRenderer, assuming Stream input:

```
renderAsStream("RenderAsText", "SimpleTextRenderer", myOptions)
```

The `options` argument passes additional information to the specified renderer, such as character encoding. The available options, their settings, and the meanings of each option / setting pair are renderer-specific. In this example, the SimpleTextRenderer uses the `myOptions` Hashtable to obtain additional information. Consult the Javadoc for each standard renderer for more information about the options available.

Example: Accepting the Default Renderer

To accept the default renderer specified by Oracle *i*FS, substitute `null` for the `rendererName` attribute. The following example shows how an application can allow the repository to select the default renderer for this Document object, assuming that this Document object can have two default renderers:

- A default text renderer
- A default XML renderer

```
renderAsStream("RenderAsText", null, myOptions)
renderAsStream("RenderAsXML",null, myOptions)
```

Introduction to Custom Renderers

The only reason for creating a custom renderer is if one of the standard renderers provided with Oracle *i*FS does not allow you to render a repository object in the format required by a particular application.

Custom renderers can be used for many purposes:

- To render a repository object in the format required by an application. For example, a Vcard needs to be rendered in the format required by the Windows Addressbook application.
- To convert documents from one MIME type to another (for example, to convert from image/jpeg to image/gif).
- To create virtual documents based on calculation and manipulation of information in the repository.
- To construct compound documents, combining data from more than one source.

Note: The sample application and code used in this chapter are taken from the "XSL Custom Renderer Sample Application Technical Brief," downloadable from the Oracle Internet File System page of the Oracle Technology Network.

How Custom Renderers Work

To develop a custom renderer you need to understand how the information flows from the custom application to the custom renderer.

Step	Area	Description/Sample
`1.`	Client	The application requests that the object be rendered by calling the appropriate renderAs*Xxx*() method. Example: `Document.renderAsStream(rendererType, rendererName, options)`
`2.`	Server	The server uses the `rendererType` and `rendererName` arguments to determine which renderer to invoke. The server invokes the renderer, passing the server-side representation of the object to be rendered and any options that were provided by the client. Example: `Renderer.renderAsStream(S_Document, options)`
`3.`	Renderer	The renderer receives the document representation and the options. The renderer then executes whatever custom code is needed to render the object in the format requested by the client.

Overview of a Renderer Application

When you plan a custom renderer application, include the following stages:

1. Write the Renderer Class.
2. Deploy the Renderer.
3. Invoke the Renderer.

Write the Renderer Class

When creating a custom renderer, you can choose from two approaches:

- If one of the existing Oracle *i*FS renderers partially meets the needs of your application, you can subclass an existing renderer. (Each of the existing Oracle *i*FS renderers implements the `oracle.ifs.server.renderers.Renderer` interface.)
- If you cannot use an existing Oracle *i*FS renderer as a starting point, you must create a custom class from scratch, directly implementing the `oracle.ifs.server.renderers.Renderer` interface.

Whichever approach you choose, writing a custom renderer means implementing the Renderer interface, either directly or indirectly. The Renderer interface defines the following two methods:

- renderAsStream()
- renderAsReader()

These renderAs*Xxxx*() methods allow the Oracle *i*FS clients or a custom application to render documents in the required format.

The syntax and arguments for each method are described below.

To write a custom renderer:

- Write a constructor.
- Write a renderAs*Xxxx*() method.

Write a Constructor

Every renderer must implement the standard constructor for a renderer. The standard constructor takes one parameter, as shown in the following table.

Parameter	Datatype	Description
`session`	S_LibrarySession	The server-side representation of the current user's LibrarySession.

Sample: Write a Constructor

```
public AirportDynamicRenderer(S_LibrarySession ifs) throws IfsException
 {
   m_IfsSession = ifs;
 }
```

Write a renderAs*Xxxx*() Method

The following table describes the parameters of the two renderAs*Xxxx*() methods:

- renderAsStream()
- renderAsReader()

Parameter	Datatype	Description
`lo`	S_LibraryObject	The object to be rendered.
`options`	Hashtable	Optional parameter. May be null. If specified, the options parameter further controls the behavior of the renderer through a set of optional name/value pairs. Commonly used to specify character encoding.

Example: Write the renderAsXxxx() Methods

The example builds up a string containing the required content and uses the StringBufferInputStream class to convert the string to an InputStream, which is the object this method returns.

The overall structure of this example is:

1. The renderAsStream() method calls renderAsString().

   The renderAsStream() method renders the specified LibraryObject as an InputStream.
2. The renderAsString() method calls renderAirport().
3. The renderAirport() method creates a String containing the required representation of the airport.

Sample Code: The renderAsStream() method

```
public InputStream renderAsStream(S_LibraryObject lo,
                                    Hashtable options)
    throws IfsException
  {
    InputStream in = null;
    String stream = renderAsString(lo, options);
    in = new ByteArrayInputStream(stream.getBytes());
    return in;
  }
```

Sample Code: The renderAsString() method

The renderAsString() method calls the renderAirport method and returns the result as a string.

```
public String renderAsString(S_LibraryObject lo,
                               Hashtable options)
    throws IfsException
  {
    String documentBody = null;
    documentBody = renderAirport(lo, options);
    return documentBody;
  }
```

Sample Code: The renderAirport() method

This method:

1. Renders an iFS object as XML with the iFS out-of-the-box SimpleXmlRenderer.
2. Obtains the XSL (stylesheet) passed in through parameter options.
3. Does the XSL transformation on the XML document from Step 1.
4. Returns the generated result.

```
public String renderAirport(S_LibraryObject lo,
                              Hashtable options)
  {

    String resultOutput = "";
    String xmlDoc = "";

    DOMParser parser = new DOMParser();

    try
    {
      //Retrieve the result from the SimpleXmlRenderer
      //This call is referencing the SimpleXMLRenderer, as defined
      //in the file AirportDefinitionPolicyBundle.xml.
      Reader reader = lo.renderAsReader("RenderXmlAirportDefinition",
                                  "AirportDefinitionXmlRenderer",
                                  null);
      BufferedReader r = new BufferedReader(reader);
      for (String nextLine = r.readLine(); nextLine != null; nextLine =
                                                      r.readLine())
        xmlDoc += nextLine;
      //Turn the XML String into an XML Document
      XMLDocument xml = ParseDocument(xmlDoc, parser);

      XMLDocument xsl = null;
      //Retrieves the XSL Style Sheet in a String format
      //from the Hashtable parameter (named options) passed in to the Renderer.
      String xslContent = (String)options.get("xsl");

      if (xslContent != null && xslContent.length() > 0 &&
                 !xslContent.toUpperCase().equals("NONE"))
      {
        try
        {
          //Turn XSL String into an XML Document
          xsl = ParseDocument(xslContent, parser);
        }
        catch (Exception e)
        {
          System.err.println("XSL : " + e.toString());
        }
      }

      if (xsl != null)
      {
        //Do the XSL Transformation.
        resultOutput = ProcessXML(xml, xsl, parser);
      }
      else
      {
        resultOutput = xmlDoc;
      }
    }
    catch (IfsException e)
    {
      resultOutput += ("<errorInRenderer type=\"IFS\">" + e.toString() +
                                                  "</errorInRenderer>");
    }
    catch (IOException e)
    {
      resultOutput += ("<errorInRenderer type=\"IO\">" + e.toString() +
                                                 "</errorInRenderer>");
    }
    //Return the result
    return resultOutput;
  }
```

Javadoc References

For more information about renderers, see the following classes in the Oracle *i*FS Javadoc.

Class	Purpose
`oracle.ifs.server.renderers.Renderer`	Interface that must be implemented for all renderers. Contains two key methods: renderAsStream() and renderAsReader().
`oracle.ifs.server.renderers.XmlRenderer`	Base class for creating a custom XML renderer. XmlRenderer converts an XML Document object into an InputStream or a Reader. Extend XmlRenderer to convert any S_LibraryObject into an XML Document representation.
`oracle.ifs.server.renderers.SimpleXmlRenderer`	Sample of a simple XML renderer based on XmlRenderer.
`oracle.ifs.server.renderers.SimpleTextRenderer`	Sample of a complete renderer for text.

Deploy the Renderer

For the protocol servers and other standard Oracle *i*FS components to access your custom renderer, the folder tree containing the class for the renderer must reside in the Oracle *i*FS CLASSPATH. Oracle *i*FS includes a special directory for this purpose. This directory, called `custom_classes`, is already in the CLASSPATH environment variable that the Oracle *i*FS server software uses.

To deploy a renderer:

1. Compile the renderer, creating a `.class` file.
2. Place the folder tree that contains the resulting `.class` file in the directory `$ORACLE_HOME/ifs/custom_classes` on the server where Oracle *i*FS is installed.

Note: The compiled Java code must be copied to the native file system of the server, not to the Oracle *i*FS repository.

Register the Renderer

The process of registering a renderer connects a class of objects with a specific renderer. You can register a renderer using any of the following facilities:

Facility	Advantages/Restrictions
Oracle *i*FS Manager	Use Oracle *i*FS Manager for simplicity and ease-of-use. Using Oracle *i*FS Manager, you can only register a renderer that exists on the same instance of Oracle *i*FS as the Oracle *i*FS Manager facility.
XML	Use XML if you prefer to register a renderer using a script, or if you need to deploy the renderer on a separate Oracle *i*FS instance.
Java	Use Java for special cases: ▪ If you need register a renderer for a specific object, instead of a class. ▪ If you need to specify multiple renderers for the same object. For example, a document might be rendered by two renderers: one for an SMB application, another for an HTTP application.

No matter which facility you use, registering a render includes the following tasks:

- Create a Policy object.
- Set the attributes of that Policy object.
- Associate the Policy object (or the PolicyProperty Bundle containing it) with a specific class or classes.

For specific information about the attributes of Policy objects, see "Using PolicyPropertyBundles to Register Renderers".

Registering a Renderer Using Oracle *i*FS Manager

To register a renderer using Oracle *i*FS Manager, follow these steps:

1. From the Oracle *i*FS Manager Object menu, choose Register.
2. From the Select Object Type window, choose Renderer Lookup.
3. From the Renderer Lookup Registry window, choose Add.
4. On the Renderer Lookup Entry window, fill in the text boxes for the attributes.
5. From the Class Association list, select the class to associate with this renderer.
6. Click OK.

Registering a Renderer Using XML

To register a renderer using XML, write an XML file to update the PolicyProperty bundle, creating a mapping between a specific class (AirportDefinition) and its associated PolicyPropertyBundle (AirportDefinitionPolicyBundle).

```
<?xml version = '1.0' standalone = 'yes'?>
<CLASSOBJECT>
       <update reftype= 'name'>AirportDefinition</update>
       <policybundle reftype='name' classname='PolicyPropertyBundle'>
             AirportDefinitionPolicyBundle
       </policybundle>
</CLASSOBJECT>
```

Sample Code: AirportDefinitionPolicyBundle

This sample code creates the PolicyPropertyBundle object, which is a collection of Property objects. For details about the attributes of Policy objects, see "Using PolicyPropertyBundles to Register Renderers".

```
<?xml version="1.0" standalone="yes"?>
<POLICYPROPERTYBUNDLE>
  <NAME> AirportDefinitionPolicyBundle </NAME>
  <PROPERTIES>
    <PROPERTY>
      <NAME> RenderXmlAirportDefinition </NAME>
      <VALUE Datatype='SystemObject' Classname='Policy' >
        <NAME> AirportDefinitionXmlRenderer </NAME>
        <IMPLEMENTATIONNAME>
          oracle.ifs.server.renderers.SimpleXmlRenderer
        </IMPLEMENTATIONNAME>
        <OPERATION> RenderXmlAirportDefinition </OPERATION>
      </VALUE>
    </PROPERTY>
    <PROPERTY>
      <NAME> CompleteDynamicRenderer </NAME>
      <VALUE Datatype='SystemObject' Classname='Policy' >
        <NAME> AirportDefinitionCompleteRenderer </NAME>
        <IMPLEMENTATIONNAME>
          ifs.sampleapps.OlivAirlines.AirportDynamicRenderer
        </IMPLEMENTATIONNAME>
        <OPERATION> CompleteDynamicRenderer </OPERATION>
      </VALUE>
    </PROPERTY>
  </PROPERTIES>
</POLICYPROPERTYBUNDLE>
```

Invoke the Renderer

This servlet calls the custom renderer and passes it the appropriate XSL stylesheet based on which client is making the request, then renders the result to the client.

Sample Code: Invoke the Renderer

```
package ifs.sampleapps.OlivAirlines;

import javax.servlet.ServletConfig;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import java.io.IOException;
import java.io.PrintWriter;
import java.io.Reader;
import java.io.BufferedReader;
import java.io.InputStream;
import java.io.InputStreamReader;
import java.util.Hashtable;

import oracle.ifs.common.IfsException;

import oracle.ifs.beans.Document;
import oracle.ifs.beans.LibraryObject;
import oracle.ifs.beans.LibraryService;
import oracle.ifs.beans.LibrarySession;
import oracle.ifs.beans.Selector;

import oracle.ifs.search.AttributeQualification;
import oracle.ifs.search.AttributeSearchSpecification;
import oracle.ifs.search.SearchClassSpecification;
import oracle.ifs.search.SearchSortSpecification;

import oracle.ifs.beans.Search;


public class iFSAirportServlet extends HttpServlet
{

  private static final boolean DEBUG = false;
  private static final int ERRORCODE = 22000;

 /**
  * Constructs the iFSAirportServlet.
  */
  public iFSAirportServlet()
  {
  }

 /**
  * The servlet container calls the init method exactly once
  * after instantiating the servlet. The init method must
  * complete successfully before the servlet can receive any requests.
  *
  * @param      ServletConfig config
  * @exception  An exception a servlet throws when it encounters difficulty.
  * @pub
  */

  public void init(ServletConfig config)
  throws ServletException
  {
    super.init(config);
  }


 /**
  * Called by the servlet container to allow the servlet to
  * respond to a request.
  * Calls the printContent method, which does the actual work.
  *
  * @param req  the ServletRequest object that contains the client's request.
  * @param res  the ServletResponse object that contains the servlet's respond.
  *
  * @exception  ServletException if an exception occurs that interferes with
  *                               the servlet's normal operation
  * @exception  java.io.IOException if an input or output exception occurs
  * @pub
  */
  public void service(HttpServletRequest request,
                      HttpServletResponse response)
  throws ServletException,
         IOException
  {
    // Retrieve Servlet's output stream
    PrintWriter out = new PrintWriter(response.getOutputStream());
    try
    {
      printContent(request, response, out);
    }
    catch (IfsException e)
    {
      out.println("<IfsException>" + e.toString() + "</IfsException>");
    }
    out.close();
  }

 /**
  * Calls the custom renderer and passes in the appropriate XSL
  * style sheet to the custom renderer based on which client is
  * making a request. Outputs the rendered result to the client.
  *
  * @param request  the ServletRequest object that contains the client's
request.
  * @param resonse  the ServletResponse object that contains the servlet's
respond.
  * @param out  for output
  * @exception  IfsException if operation fails.
  * @exception  IOException if an input or output exception occurs.
  * @pub
  */
  private void printContent(HttpServletRequest request,
                            HttpServletResponse response,
                            PrintWriter out) throws IOException,
                                                    IfsException
  {
    // Retrieve User-Agent to know which kind of client is making the request.
    String userAgent = request.getHeader("User-Agent");

    LibraryService service = new LibraryService();

    String userName = request.getParameter("userName");
    String passWord = request.getParameter("passWord");
    String serviceName = request.getParameter("serviceName");
    LibrarySession  ifs = service.connect(userName, passWord, serviceName);
    // Finds the iFS object we are doing to render with a Selector.
    Selector mySelector = new Selector(ifs);
    // Select the Airportdefinition class based on its attribute: AIRPORTCODE.
    mySelector.setSearchClassname("AIRPORTDEFINITION");

    // The airport code is passed in as a parameter provide along with the URL.
    String code  = request.getParameter("code");
    mySelector.setSearchSelection("AIRPORTCODE = '" + code + "'");
    for (int i=0; i<mySelector.getItemCount(); i++)
    {
      LibraryObject lo = mySelector.getItems(i);
      String contentType = "";

      Hashtable h = new Hashtable();
      // Check if a given string ("HANDHTTP" here) matchs any substring
      // of the User-Agent parameter passed in, case insensative.
      // If it matches, the corresponding style sheet is read from iFS,
      // and put in an Hashtable, to be passed in to the renderer.
      if (userAgent.toUpperCase().indexOf("HANDHTTP") > -1)
      {
         h.put("xsl", getStyleSheetContent(ifs, "apHTML.xsl"));
         contentType = "text/html";
      }
      else if (userAgent.toUpperCase().indexOf("MOZILLA") > -1)
      {
         h.put("xsl", getStyleSheetContent(ifs, "apHTML.xsl"));
         contentType = "text/html";
      }
      else if (userAgent.toUpperCase().indexOf("UP") > -1)
      {
         h.put("xsl", getStyleSheetContent(ifs, "apWAP.xsl"));
         contentType = "text/x-wap.wml";
      }
      else if (userAgent.toUpperCase().indexOf("NOKIA") > -1)
      {
         h.put("xsl", getStyleSheetContent(ifs, "apWAP.xsl"));
         contentType = "text/x-wap.wml";
      }
      else if (userAgent.toUpperCase().indexOf("MOTOROLA") > -1)
      {
         h.put("xsl", getStyleSheetContent(ifs, "apVox.xsl"));
         contentType = "text/html";
      }
      else
      {
         h.put("xsl", "none");
      }

      response.setContentType(contentType);
     // Calls the custom renderer. Pass in the Hashtable
     // The custom renderer is registered in the
     // AirportDefinitionPolicyBundle.xml file.
  Reader reader = lo.renderAsReader("CompleteDynamicRenderer",
"AirportDefinitionCompleteRenderer", h);
     // Reads results from the Reader, and prints to the Servlet output.
     printAirport(reader, out);
    } //end for loop
  }


 /**
 * This method seaches and gets the content of an iFS document, the XSL style
 * sheet that will be passed to the custom renderer, based on its file name.
 **/
 private static String getStyleSheetContent(LibrarySession ifs, String xslName)
   throws IfsException
  {
    String retString = "";

    String className[] = {"DOCUMENT"};
    SearchClassSpecification scs = new SearchClassSpecification(className);
    scs.addResultClass("DOCUMENT");
    AttributeQualification aq1 = new AttributeQualification();
    aq1.setAttribute("DOCUMENT", "NAME");
    aq1.setOperatorType(AttributeQualification.LIKE);
    aq1.setValue(xslName);
    SearchSortSpecification ss = new SearchSortSpecification();
    ss.add("NAME" , SearchSortSpecification.ASCENDING);
    AttributeSearchSpecification ass = new AttributeSearchSpecification();
    ass.setSearchClassSpecification(scs);
    ass.setSearchQualification(aq1);
    ass.setSearchSortSpecification(ss);
    Search srch = new Search(ifs, ass);
    srch.open();
    try
    {  while (true)
       {
          LibraryObject lo = srch.next().getLibraryObject();
          Document d = (Document)lo;
          InputStream is = d.getContentStream();
          BufferedReader br = new BufferedReader(new InputStreamReader(is));
          for (String nextLine = br.readLine(); nextLine != null; nextLine =
br.readLine())
          {
            retString += nextLine;
          }
          br.close();
        }
    }
    catch (IfsException e)
    {
      if (e.getErrorCode() == ERRORCODE)
      {    [Is this set of braces needed????}
       }
      else
      {
        throw e;
      }
    }
    catch (IOException ioe)
    {
       System.err.println("IOException reading XSL : " + ioe.toString());
    }
    srch.close();
    return retString;
  }

 /**
  * Prints out the renderer output to the client.
  *
  * @param reader   the renderer output passed in
  * @param out      for output
  */
  public static void printAirport(Reader reader,
                                  PrintWriter out)
  throws IOException
  {
     // Dumps the reader on the output.
     BufferedReader r = new BufferedReader(reader);
     for (String nextLine = r.readLine(); nextLine != null; nextLine =
r.readLine())
     {
        out.println(nextLine);
     }
   }


}
```

Output from the Custom Renderer

To run the servlet:

1. Open a web browser.
2. Type the following command in the Location window:

   Http://*machineName:portNumber*/XSLRenderer?code=LAX&userName=*yourUserName*&passWord=*yourPassWord*&serviceName=*yourServiceName*

   where:

   `machineName` is the server name where the iFS is running.

   `portNumber` is the port where the Java Web Server is running. Get the port number by typing `http://machineName:9090`.

   `yourUserName` is the user you created during this example.

   `yourPassWord` is the password for this user.

   `serviceName` is the Service Name of iFS. The default for the Service Name is ServerManager.

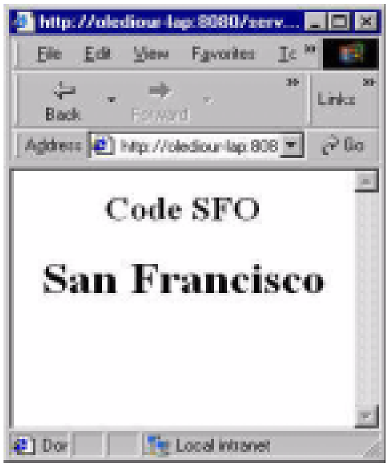
You should see LAX and Los Angeles in the browser. You can change the code to SEA or SFO. For example:

```
http://bsmith-sun:80/XSLRenderer?code=LAX&userName=gking&pa
ssWord=ifs&serviceName=ServerManager
```

Access the Servlet from Different Devices

Try entering the command from the previous step into different devices to see what the output looks like.

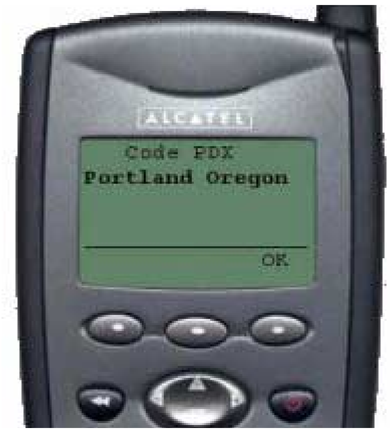
Figure 6–1 Output of AirportDefinition as seen on a desktop HTML Browser


http://olediour-lap:8080/serv...
File Edit View Favorites Tc
Back Forward Links
Address http://olediour-lap:808 Go
Done Local intranet

Code SFO

San Francisco

Figure 6–2 Output of AirportDefinition as seen on a cellphone



Code PDX
Portland Oregon

Figure 6–3 Output of AirportDefinition as seen on the HTML browser of a personal digital assistant.

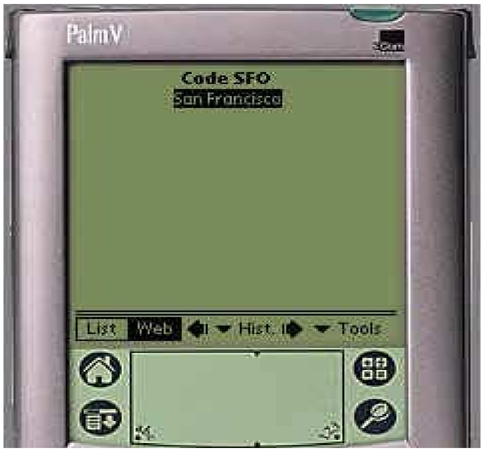
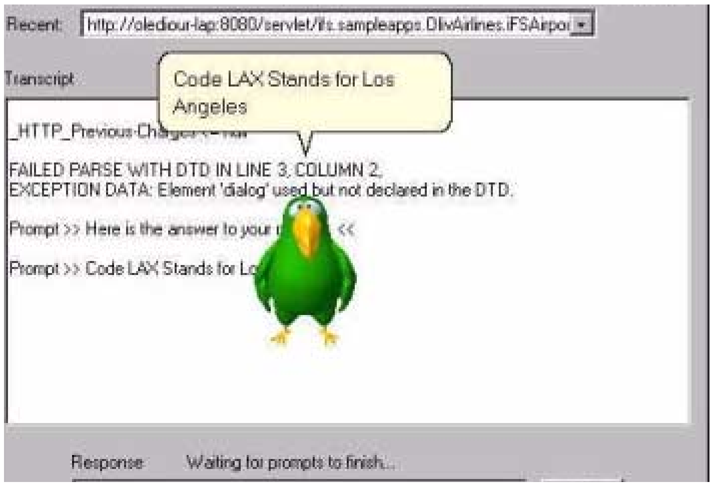


Figure 6–4 Output of AirportDefinition as displayed through a Voice Simulator.



```
http://olediour-lap:8080/servlet/ifs.sampleapps.OlivAirlines.iFSAirpo
_HTTP_Previous-Ch
FAILED PARSE WITH DTD IN LINE 3, COLUMN 2,
EXCEPTION DATA: Element 'dialog' used but not declared in the DTD.
Prompt >> Here is the answer to your                <<
Prompt >> Code LAX Stands for Lo
Waiting for prompts to finish...
```

7 Using JSPs

This chapter covers the following topics:

- Using Java Server Pages to Display Documents
- Implementing an Application Using a JSP
- Running the Sample Insurance Form Application
- Sample Files for the Insurance Form Application

Note: Do not confuse Java Server Pages, commonly known as "JSPs," with the Oracle Corporation product called Java Stored Procedures. In the context of this chapter, "JSPs" refers to "Java Server Pages."

Using Java Server Pages to Display Documents

One of the advantages of storing documents in the Oracle Internet File System is that users can display dynamic data via the Web simply by entering a URL. For example, in an insurance form application, a user would be able to view an insurance form on the Web.

Java Server Pages (JSPs) are the recommended manner of creating web-based applications based on Oracle *i*FS. (In fact, the Oracle *i*FS Web interface uses JSPs.) JSPs display information on the Web by generating HTML documents to call Java Beans. Although JSPs provide a way for users to view and manipulate documents, providing a functionality parallel to that of renderers, using a JSP is not technically "rendering" because it does not involve calling a specific renderer.

Note: You can use either the Java Web Server or the Apache Web Server with Oracle *i*FS. The Java Web Server included with Oracle *i*FS 1.1 supports the 1.0 version of the JSP specification. Developers are advised to avoid the use of tag extensions defined by the 0.92 specification.

Preparing to Use JSPs

An insurance form application is used to demonstrate how to use JSPs. Assume that the following preparations have been completed so that there are custom InsuranceForm objects stored in the repository, ready to be displayed using a JSP:

1. Using XML, create a type definition file, `CreateInsuranceForm.xml`, to define the custom attributes of the insurance form. (Creating this type definition file in XML is the equivalent of subclassing the Document class in Java.)
2. Using Java, create an instance class Bean, `InsuranceFormObject.java`, to provide standard getter and setter methods for the custom attributes defined in the type file. (This step is recommended, but not required. The instance class Bean file is shown in Chapter 4, "Creating Custom Classes".)
3. Using XML, create the actual insurance form documents, claim1.xml and claim2.xml, each one created as an instance of the InsuranceForm class. Load the document instance files into Oracle *i*FS using either the Web interface or the Windows interface. The insurance forms will be parsed automatically by the SimpleXmlParser.

Implementing an Application Using a JSP

An application that uses a JSP to display information consists of four components:

Component	Purpose	Files/Utility Needed	Example
Login files	Authenticate the user.	Login JSP	`Sample Code: login.jsp`
		Corresponding Java Bean	`Sample Code: InsuranceLogin.java`
Logout files	Close the session.	Logout JSP	`Sample Code: logout.jsp`
Application files	Display information stored in Oracle *i*FS using the Web.	Application JSP	`Sample Code: InsuranceForm.jsp`
		Corresponding Java Bean	`Sample Code: InsuranceBean.java`
Registration using Oracle *i*FS Manager	Connect a document type with the JSP used to display it.	Oracle *i*FS Manager	`Registering a Java Server Page Using Oracle iFS Manager`

Login/Logout Files

Each application requires a set of files to provide basic security functions:

- User Login and Validation
- User Logout

The sample code for these functions, login.jsp and logout.jsp, provides a reliable starting point for a login / logout mechanism.

For consistent behavior with multiple protocols and sharing login information with the Oracle *i*FS Web interface, we recommend using the basic approach to login / logout demonstrated in the sample code. Specifically, we recommend that when writing a JSP login facility, you implement the Oracle *i*FS Java API package `oracle.ifs.adk.security.IfsHttpLogin.`

User Login and Validation

For an example of a login JSP and its associated Java Bean, see:

- "Sample Code: login.jsp"
- "Sample Code: InsuranceLogin.java"
  InsuranceLogin.java implements the public interface `IfsHttpLogin`.

These sample applications store the minimum login and session information needed by the system to carry user authentication from page to page. Depending on the requirements of your application, you may decide to create a more complex login to store user variables or more extensive session information.

User Logout

For an example of a simple logout JSP, see "Sample Code: logout.jsp".

Application Files

The real work of an application is done by one or more JSPs and their corresponding Java Beans:

- The JSP calls the Java Bean and creates an HTML page to display the information from the database.

  For a sample JSP file, see "Sample Code: InsuranceForm.jsp". This sample shows a simple user interface; your application may require a more sophisticated user interface.

- The Java Bean retrieves and manipulates information from the database. The Java Bean contains the business application logic. For example, if your application requires that all invoices with a total greater than $10,000 be approved by a specific manager, that logic would go in the Java Bean.

  For a sample Java Bean file, see "Sample Code: InsuranceBean.java".

Compiling Java Beans and JSPs

You must compile the Java Bean and then the JSP before Oracle *i*FS can call the JSP. Depending on your development environment, the JSP may be automatically compiled at runtime the first time it is invoked.

Deploying Java Beans and JSPs

All compiled Java Beans must be stored in the directory `$ORACLE_HOME/ifs/custom_classes` on the server where Oracle *i*FS is installed.

The Oracle *i*FS product comes with a starting-point directory structure. One of the folders in that structure is reserved for holding JSPs: `$ORACLE_HOME/ifs/jsp-bin`. To have your JSP compile and execute properly in Oracle *i*FS, you must store the JSP in this folder. JSPs stored elsewhere will not be executed by the Oracle *i*FS JSP compiler.

Registering a JSP

Registering a JSP allows the JSP to be used as the default HTML renderer for the specified class.When a user clicks on a plain XML document, the system automatically calls the SimpleXmlRenderer supplied with Oracle *i*FS. When a user clicks on a custom document for which you have provided a JSP, Oracle *i*FS needs to know which JSP to use to display documents of this type. To provide this information to Oracle *i*FS, you must register a JSP before it can be used.

You can use Oracle *i*FS Manager or XML to register a JSP. Once a JSP has been registered, Oracle *i*FS automatically invokes that JSP when a user requests a document of a given class and MIME type. Behind the scenes, registering a JSP adds an entry to the existing JSPlookup PropertyBundle, associating a JSP with a specific class and MIME type. For an example, see "Registering a Java Server Page Using Oracle iFS Manager".

Note: Using the Jsplookup PropertyBundle mechanism will meet the requirements of most applications. If your application has complicated requirements in this area, you are free to use a custom method to trigger the use of specific code when a user accesses custom documents.

The Jsplookup PropertyBundle

To associate a JSP with a specific class, Oracle *i*FS provides a lookup table mechanism called the "Jsplookup PropertyBundle." The Jsplookup PropertyBundle is automatically created during installation.

The Jsplookup PropertyBundle maps a specific class and MIME type to the corresponding JSP. In the best case, both document class and document MIME type are available. Depending on the specific document, however, sometimes only document class or MIME type are available. The Jsplookup PropertyBundle determines which JSP should be used with which class based on all available information about the class, assessed in this sequence:

1. Combination of document class and document MIME type.
2. Document class only.
3. Document MIME type only.

Web Site Security Using HTTP Authentication

HTTP authentication is used to control access to web sites. For example:

- You may want to limit access to all of your web site.
- You may want only specific users to be able to access certain parts of the web site.

Assume, for example, that you have a web site with a "Members Only" section. You would want everyone to have access to the web site, but only people with a special password have access to the "Members Only" section.

The HTTP authentication mechanism causes the browser to pop up a dialog window. This dialog window accepts the following input from the user:

- User name
- Password

Once a valid value has been supplied, this user name/password information is kept with the current session, which allows this user access to all appropriate documents, based on the security provided by the associated ACLs. Note that this user's user name/password information is stored by the browser. Thus, if the user invokes additional instances of the browser, the authentication is also valid for these instances. Until the user exits from all instances of the browser, these access rights are not disabled. This behavior is browser-specific. For example, if a user authenticates using Netscape Communicator, that user does not need to authenticate for any more instances of Netscape Communicator, but will need to provide authentication for an instance of Internet Explorer.

In order to use HTTP authentication, you must substitute your own "index.html" for the default "index.html" in the root directory, so users will go to your web site rather than the Oracle *i*FS Web interface. When users log in through the Web interface, the Web interface authentication takes precedence over HTTP authentication.

Implementing HTTP Authentication

Oracle *i*FS implements HTTP authentication using Access Control Lists (ACLs). Assume that you have a web site where you want to specify the following access for the "Members Only" section:

- Everyone should be able to see that the "Members Only" section exists.
- Read access to the "Members Only" section is restricted to specific users.

To access this restricted section, the user needs to click a "Members Only" button (`members.gif`) displayed on the Home page.

To implement HTTP authentication for this web site according the scenario described above, follow this process.

1. The default `index.html` file in the root (/) directory points to the Oracle *i*FS Web interface. As part of creating your web site, create a file called `index.html` and replace the `index.html` file in the root directory with the file that points to your web site.
2. As another part of creating your web site, create an HTML document to serve as the gateway to the "Members Only" section.

   This HTML page will serve as the HREF for `members.gif`.
3. Log in using an administration-enabled account, and create a new ACL to handle HTTP authorization.

   Assume this new ACL is named "HTTP_Auth." Assign the following rights:
   - Guest - Discover access only.
   - Qualified user - Read access.
4. Log in using an administration-enabled account, and apply the HTTP_Auth ACL to the `index.html` document created in Step 1.
5. Exit the browser.

Now when a user enters the web site and clicks the "Members Only" button, the authentication dialog will be displayed. If the user provides the correct user name and password, that user will gain access to the "Members Only" section.

Running the Sample Insurance Form Application

The Insurance Form application includes three JSPs:

- "Sample Code: login.jsp" allows a user to log in to Oracle *i*FS.
- "Sample Code: logout.jsp" allows a user to log out of Oracle *i*FS.
- "Sample Code: InsuranceForm.jsp" provides the application logic.

Note: This example includes the recommended mechanism for providing a login facility to Oracle *i*FS, which is implementing the `IfsHttpLogin` interface.

For a brief description of each sample file, see "Sample Files for the Insurance Form Application".

Create the Insurance Form Application

1. Create these directories:
   - `public/examples/insuranceApp`
   - `public/examples/insuranceApp/src`
   - `public/examples/insuranceApp/claims`
2. Compile these Java programs:
   - `InsuranceLogin.java`
   - `InsuranceBean.java`

   Place the folder tree containing the resulting `.class` file in the directory `$ORACLE_HOME/ifs/custom_classes` on the server where Oracle *i*FS is installed.
3. Use FTP to put the Java source code into `/public/examples/insuranceApp/src`.
4. Use FTP to put `index.html` into `/public/examples/insuranceApp`.
5. Use FTP to put `CreateInsuranceForm.xml` into any convenient directory.
6. Using Oracle *i*FS Manager, register the `InsuranceForm.jsp`.
7. Use FTP to put the following two document files representing claims into `/public/examples/insuranceApp/claims`:
   - `claim1.xml`
   - `claim2.xml`
8. Use FTP to put the three JSPs into `/ifs/jsp-bin`:
   - `login.jsp`
   - `logout.jsp`
   - `InsuranceForm.jsp`

Run the Insurance Form Application

1. Start the Web Server.
2. Enter the following URL into your favorite web browser:

   ```
   http://myIfsServer/public/examples/insuranceApp/src/login.jsp
   ```

3. Log in to Oracle *i*FS.
4. To view the claims, log in and click on the claim files:
   - Juana Angeles
   - Kevin Chu
5. To logout, click on `logout.jsp`.

Requirements for Running the Examples

To run these examples, observe the following requirements:

- Compile all `.java` files.
- To compile, include the following .jar files in the `CLASSPATH` environment variable:
  - `repos.jar`
  - `adk.jar`
- Be sure the JSPs are stored in `/ifs/jsp-bin`.

Sample Files for the Insurance Form Application

This section includes sample code for running the Insurance Form application. The files included are:

Name	Description
`index.html`	An HTML file that is automatically invoked when a user's URL navigates to a folder.
`CreateInsuranceForm.xml`	An XML file that creates a custom document type (a subclass of the Document class).
`claim1.xml` `claim2.xml`	Two XML insurance claim files that create instances of the InsuranceForm class.
`login.jsp`	A JSP to allow user login and validation. Calls `InsuranceLogin.java.`
`InsuranceLogin.java`	A Java Bean called by `login.jsp.`
`logout.jsp`	A JSP to allow user logout.
`RegisterJSP.xml`	An XML file to register a JSP, thus associating the JSP with a specific class and MIME type. Alternatively, you can use Oracle *i*FS Manager to register a JSP.
`InsuranceForm.jsp`	A JSP to display application information retrieved by `InsuranceBean.java`.
`InsuranceBean.java`	A Java Bean called by `InsuranceForm.jsp` to retrieve attributes of a specific insurance form.

Sample Code: index.html

This `index.html` file is automatically invoked when a user, using a URL, navigates to the folder where this file resides.

```
<HTML>
<!-- index.html -->
<HEAD>
<META HTTP-EQUIV=REFRESH CONTENT="0;
  URL=/public/examples/insuranceApp/src/login.jsp">
</META>
</HEAD>
<BODY>
</BODY>
</HTML>
```

Sample Code: CreateInsuranceForm.xml

This `CreateInsuranceForm.xml` file creates a custom document type with two custom attributes, ClaimNumber and ClaimType.

```
<?xml version = '1.0' standalone = 'yes'?>
<!-- CreateInsuranceForm.xml -->
<ClassObject>
   <Name>InsuranceForm</Name>
   <Superclass Reftype ="name">Document</Superclass>
   <Attributes>
      <Attribute>
          <Name>ClaimNumber</Name>
          <DataType>Long</DataType>
      </Attribute>
      <Attribute>
          <Name>ClaimType</Name>
          <DataType>String</DataType>
          <DataLength>50</DataLength>
      </Attribute>
   </Attributes>
</ClassObject>
```

Sample Code: claim1.xml, claim2.xml

These two files, `claim1.xml` and `claim2.xml`, are the document files that create two specific instances of the InsuranceForm class.

```
<?xml version = '1.0' standalone = 'yes'?>
<!-- claim1.xml -->
<InsuranceForm>
   <Name>Juana Angeles</Name>
   <ClaimNumber>35093</ClaimNumber>
   <ClaimType>Car Accident</ClaimType>
   <FolderPath>/public/examples/insuranceApp/claims</FolderPath>
</InsuranceForm>

<?xml version = '1.0' standalone = 'yes'?>
<!-- claim2.xml -->
<InsuranceForm>
   <Name>Kevin Chu</Name>
   <ClaimNumber>41111</ClaimNumber>
   <ClaimType>Car Accident</ClaimType>
   <FolderPath>/public/examples/insuranceApp/claims</FolderPath>
</InsuranceForm>
```

Sample Code: login.jsp

This `login.jsp` file is a JSP that provides for user login and validation. It calls the `InsuranceLogin.java` file. Note that "IfsHttpLogin" is the default login Bean name used for the Oracle *i*FS Web interface.

```
<%@ page import = "ifsdevkit.sampleapps.insurance.InsuranceLogin" %>
<%@ page import = "oracle.ifs.adk.security.IfsHttpLogin" %>

<html><head>

<jsp:useBean id="inslogin" scope="session"
class="ifsdevkit.sampleapps.insurance.InsuranceLogin" />
<jsp:setProperty name="inslogin" property="*"/>

<%
  String REDIRECT_PATH = "/public/examples/insuranceApp/claims";
  boolean loggedIn = false;
  if (inslogin.getSession() != null && inslogin.getResolver() != null)
  {
    // Use existing insurance login
    loggedIn = true;
  }
  else
  {
    // No existing insurance login
    IfsHttpLogin login = (IfsHttpLogin)
request.getSession(true).getValue("IfsHttpLogin");
    if (login != null && login.getSession() != null && login.getResolver() !=
null)
    {
      // Use existing IfsHttpLogin login
      inslogin.init(login.getSession(), login.getResolver());
      loggedIn = true;
    }
  }
  if (!loggedIn)
  {
    String username = request.getParameter("username");
    String password = request.getParameter("password");
    if (username != null && password != null)
    {
      // Login using username/password
      try
      {
        inslogin.init(username, password, "IfsDefault");
        request.getSession(true).putValue("IfsHttpLogin", inslogin);
        loggedIn = true;
      }
      catch (Exception e)
      {
%>
<SCRIPT LANGUAGE="JavaScript1.2">
        alert("The username or Password was not valid, please try again.");
</SCRIPT>
<%
      }
    }
  }
  if (loggedIn)
  {
    // Redirect to the directory where the claim files reside
    response.sendRedirect(REDIRECT_PATH);
  }
  else
  {
%>
<title>Insurance Demo App Login</title>
<meta http-equiv="Content-Type" content="text/html; charset=iso-8859-1">
</head>
<body bgcolor="#FFFFFF">
  <form METHOD=POST NAME="loginform" ACTION="login.jsp">
    <table>
      <tr>
        <td><b>Username:</b></td>
        <td><input type="text" name="username" value=""></td>
      </tr>
      <tr>
        <td><b>Password:</b></td>
        <td><input type="password" name="password" value=""></td>
      </tr>
      <tr>
        <td>&nbsp;</td>
        <td>&nbsp;</td>
      </tr>
      <tr>
        <td>
          <input type="submit" value="Log in">
        </td>
        <td>
          <input type="reset" value="Reset">
        </td>
      </tr>
    </table>
  </form>
</body>
</html>

<% } %>
```

Sample Code: InsuranceLogin.java

This sample file, `InsuranceLogin.java`, creates the Java Bean that is called by `login.jsp`, the corresponding JSP. This Java Bean implements the standard Oracle *i*FS login interface, `IfsHttpLogin`.

```
/* --InsuranceLogin.java-- */
package ifsdevkit.sampleapps.insurance;

import java.util.Locale;
import javax.servlet.http.HttpSessionBindingEvent;
import oracle.ifs.beans.DirectoryUser;
import oracle.ifs.beans.FolderPathResolver;
import oracle.ifs.beans.LibrarySession;
import oracle.ifs.beans.LibraryService;
import oracle.ifs.common.CleartextCredential;
import oracle.ifs.common.ConnectOptions;
import oracle.ifs.common.IfsException;
import oracle.ifs.adk.security.IfsHttpLogin;

/**
 * The login bean for the Insurance demo app.
 * <p>
 * This class provide the login info. The class implements the
 * <code>IfsHttpLogin</code> interface so it can share login data with other
 * login beans.
 *
 * @see IfsHttpLogin
 */

public class InsuranceLogin implements IfsHttpLogin
{
  /**
   * The <code>LibrarySession</code>.
   */
  private LibrarySession m_session;

  /**
   * The <code>FolderPathResolver</code>.
   */
  private FolderPathResolver m_resolver;

  /**
   * Default constructor required by the jsp spec for the USEBEAN tag
   *
   * @exception IfsException
   */
  public InsuranceLogin()
  throws IfsException
  {
  }

  /**
   * Make a connection to iFS
   *
   * @param username  The username to be used for login.
   *
   * @param password  The password to be used for login.
   *
   * @param server    The server to be used for login.
   *
   * @exception IfsException if operation failed.
   */
  public void init(String username, String password, String serviceName)
  throws IfsException
  {
    LibraryService service = new LibraryService();

    CleartextCredential me = new CleartextCredential(username, password);
    ConnectOptions connection = new ConnectOptions();
    connection.setLocale(Locale.getDefault());
    connection.setServiceName(serviceName);
    m_session = service.connect(me, connection);

    m_resolver = new FolderPathResolver(m_session);

    m_resolver.setRootFolder();

    DirectoryUser user = m_session.getDirectoryUser();
    if (user.isAdminEnabled())
      m_session.setAdministrationMode(true);
  }

  /**
   * Initialize the login bean.
   * <p>
   * The default constructor does not set the necessary fields so it needs
   * to be set instantiation.
   *
   * @param session   The <code>LibrarySession</code> object.
   *
   * @param resolver  The <code>FolderPathResolver</code> object.
   *
   */
  public void init(LibrarySession session, FolderPathResolver resolver)
  {
    m_session = session;
    m_resolver = resolver;
  }

  /**
```

```
     * Return the login's session object.
     *
     * @return The <code>LibrarySession</code> object.
     */
    public LibrarySession getSession()
    {
      return m_session;
    }

    /**
     * Return the login's path resolver.
     *
     * @return The <code>FolderPathResolver</code> object.
     */
    public FolderPathResolver getResolver()
    {
      return m_resolver;
    }

    /**
     * Called when this object is bound to the HTTP session object.
     *
     * @param event  The event when the object is bound to the Http session.
     */
    public void valueBound(HttpSessionBindingEvent event)
    {
      // do nothing
    }

    /**
     * Called when this object is unbound from the HTTP session object.
     *
     * @param event  The event when the object is unbound to the Http session.
     */
    public void valueUnbound(HttpSessionBindingEvent event)
    {
      m_resolver = null;
      try
      {
        if (m_session != null)
        {
          m_session.disconnect();
        }
      }
      catch (IfsException e)
      {
        e.printStackTrace();
      }
      finally
      {
        m_session = null;  // release the resources
      }
    }

  }
```

Sample Code: logout.jsp

This sample file, `logout.jsp`, is a JSP that provides for user logout and a graceful exit from the program.

```
<%@ page import = "ifsdevkit.sampleapps.insurance.InsuranceLogin" %>
<%@ page import = "oracle.ifs.adk.security.IfsHttpLogin" %>

<html><head>

<%
  Object login = request.getSession(true).getValue("inslogin");
  if (login != null)
  {
    // Remove insurance login
    request.getSession(true).removeValue("inslogin");
    Object ifsLogin = request.getSession(true).getValue("IfsHttpLogin");
    if (ifsLogin != null && ifsLogin == login)
    { // Only remove IfsHttpLogin if it is the same login bean
      // Remove IfsHttpLogin login
      request.getSession(true).removeValue("IfsHttpLogin");
    }
  }
%>
<title>Insurance Demo App Logout</title>
<meta http-equiv="Content-Type" content="text/html; charset=iso-8859-1">
</head>
<body bgcolor="#FFFFFF">
  <form METHOD=POST NAME="logout" ACTION="login.jsp">
    <table>
      <tr>
        <td><h3>You are now logged out.</h3></td>
      </tr>
      <tr>
```

```
        <td>&nbsp;</td>
      </tr>
      <tr>
        <td>
          <input type="submit" value="Log in">
        </td>
      </tr>
    </table>
  </form>
</body>
</html>
```

Registering a Java Server Page Using Oracle *i*FS Manager

To register a Java Server Page using Oracle *i*FS Manager, follow these steps:

1. From the Oracle *i*FS Manager Object menu, choose Register.
2. From the Select Object Type window, choose Java Server Page (JSP) Lookup.
3. From the Java Server Page (JSP) Lookup Registry window, choose Add.
4. In the Java Server Page (JSP) Lookup Entry window, specify the attributes according the JSP Attributes table.

To register a JSP, specify the following three attributes:

Attribute	Description	Examples
Classname	Name of the custom class that uses this JSP. Case-sensitive.	`INSURANCEFORM`
Mimetype	MIME type and subtype of files this JSP can render, in the format: `MyMIMEType1/MyMIMEType2`	`text/HTML` `video/quicktime`
	`MyMIMEType1` specifies a main MIME type, such as "text"; `MyMIMEType2` specifies a MIME subtype, such as "HTML".	To specify that all MIME types can be rendered, use */*: `INSURANCEFORM.*/*`
JSP pathname	Location of the compiled JSP.	`/ifs/jsp-bin/InsuranceForm.jsp`

5. Click OK.

Registering a Java Server Page Using XML

This sample file, `RegisterJSP.xml`, registers a JSP, associating the JSP with a specific classname and MIME type.

To register a JSP using XML, write an XML file to update the JspLookup PropertyBundle, adding a mapping between a specific classname and MIME type (`INSURANCEFORM.*/*`) and its associated JSP (`InsuranceForm.jsp`).

```
<?xml version="1.0" standalone="yes"?>
<!--RegisterJSP.xml-->
<PROPERTYBUNDLE>
   <UPDATE RefType="valuedefault">JspLookup</UPDATE>
   <PROPERTIES>
      <PROPERTY ACTION="add">
         <NAME>INSURANCEFORM.*/*</NAME>
         <VALUE DataType="String">/ifs/jsp-bin/InsuranceForm.jsp</VALUE>
      </PROPERTY>
   </PROPERTIES>
</PROPERTYBUNDLE>
```

Sample Code: InsuranceForm.jsp

This sample file, `InsuranceForm.jsp`, includes the following:

- HTML code to govern the display of the insurance form.
- A call to the Java Bean `InsuranceBean.java`, which provides the information from the insurance form stored in the repository.

```
<HTML>
<!-- InsuranceForm.jsp-->
<%@ page import="ifsdevkit.sampleapps.insurance.InsuranceBean" %>
<%@ page import="oracle.ifs.beans.LibrarySession" %>
<%@ page import="oracle.ifs.beans.FolderPathResolver" %>
<%@ page import="oracle.ifs.adk.security.IfsHttpLogin" %>

<jsp:useBean id="ibean" scope="session"
class="ifsdevkit.sampleapps.insurance.InsuranceBean" />
<jsp:useBean id="inslogin" scope="session"
class="ifsdevkit.sampleapps.insurance.InsuranceLogin" />

<HEAD>
    <TITLE>Oracle iFS</TITLE>
</HEAD>
<BODY>
<%
```

```
  LibrarySession sess = null;
  FolderPathResolver resolver = null;
  if (inslogin != null && inslogin.getSession() != null &&
inslogin.getResolver() != null)
  {
    sess = inslogin.getSession();
    resolver = inslogin.getResolver();
  }
  else // Not logged in the Insurance App but may be logged in other
application.
  {
    IfsHttpLogin login =
(IfsHttpLogin)request.getSession(true).getValue("IfsHttpLogin");
    if (login != null && login.getSession() != null && login.getResolver() !=
null)
    {
      sess = login.getSession();
      resolver = login.getResolver();
    }
  }
  String path = request.getParameter("path");
  if (path == null || sess == null || resolver == null)
  {

response.sendRedirect("/ifs/jsp-bin/ifsdevkit/sampleapps/insurance/login.jsp");
  }
  else
  {
    ibean.init(sess, resolver, path);
%>
<TABLE>
<TR>
<TH ALIGN="left">Name:</TH><TD ALIGN="left"><%= ibean.getName() %></TD>
</TR>
<TR>
<TH ALIGN="left">ClaimType: </TH><TD ALIGN="left"> <%= ibean.getClaimType() %>
</TD>
</TR>
<TR>
<TH ALIGN="left">ClaimNumber: </TH><TD ALIGN="left"> <%= ibean.getClaimNumber()
%> </TD>
</TR>
</TABLE>

</BODY>
</HTML>

<%
  }
%>
```

Sample Code: InsuranceBean.java

This sample code, `InsuranceBean.java`, creates the JavaBean that is called by `InsuranceForm.jsp`, the corresponding JSP. This JavaBean retrieves three attributes of the insurance form: Name, ClaimNumber, and ClaimType.

```
/*---InsuranceBean.java---*/
package ifsdevkit.sampleapps.insurance;

import oracle.ifs.beans.ClassObject;
import oracle.ifs.beans.DirectoryUser;
import oracle.ifs.beans.FolderPathResolver;
import oracle.ifs.beans.LibrarySession;
import oracle.ifs.beans.PublicObject;
import oracle.ifs.common.AttributeValue;
import oracle.ifs.common.IfsException;

public class InsuranceBean
{
  /**
   * The name for the claim.
   */
  protected String m_name;

  /**
   * The type of the claim.
   */
  protected String m_claimType;

  /**
   * The claim number.
   */
  protected Long   m_claimNumber;

  /**
   * Constructor
   */
  public InsuranceBean()
  {
  }

  /**
   * Initialize the bean and populate the necessary fields.
   *
   * @param session   The <code>LibrarySession</code> object.
   *
   * @param resolver  The <code>FolderPathResolver</code> object.
   *
   * @param path  The path to the insurance object.
   *
   * @exception IfsException Thrown if operation failed.
   */
  public void init(LibrarySession session, FolderPathResolver resolver, String
path)
  throws IfsException
  {
    try
    {
      PublicObject insuranceObj = resolver.findPublicObjectByPath(path);
      ClassObject co = insuranceObj.getClassObject();

      m_name = insuranceObj.getName();
      AttributeValue av = insuranceObj.getAttribute("CLAIMTYPE");
      if (!av.isNullValue())
      {
        m_claimType = av.getString(session);
      }
      av = insuranceObj.getAttribute("CLAIMNUMBER");
      if (!av.isNullValue())
      {
        m_claimNumber = new Long(av.getLong(session));
      }
    }
    catch (IfsException e)
    {
      e.printStackTrace();
      throw e;
    }
  }

  /**
   * Return name for the claim.
   *
   * @return The claim name.
```

```
     */
    public String getName()
    {
      return m_name;
    }

    /**
     * Return the claim type.
     *
     * @return The claim type in a <code>String</code>.
     */
    public String getClaimType()
    {
      return m_claimType;
    }

    /**
     * Return the claim number.
     *
     * @return The claim number as a <code>Long</code>.
     */
    public Long getClaimNumber()
    {
      return m_claimNumber;
    }
  }
```

8 Using Agents

This chapter covers the following topics:

What Is an Agent?

An agent is a Java program used to automate a task. More specifically, an agent lets an application respond to specific actions taken within the Oracle Internet File System environment. For example, an agent may respond when an instance of a certain document class is inserted, updated, or deleted in the repository.

One common use of agents is to provide notifications. For example, application design may require that when anyone inserts a document in a certain folder, a notification should be sent to a specific person, reporting that fact. An example of using an agent to provide notification is to implement the following business rule: "Whenever a purchase order is placed in the "Ready for Approval" folder, a message should be sent to the vice president of the division, who approves all purchase orders." To implement this functionality in Oracle *i*FS, write an agent.

What Triggers an Agent's Action?

You can think of agents as belonging to certain categories, depending on what triggers their actions. Agents can be triggered by time, events, or both. Because most custom agents are event-based agents, the remainder of this chapter will focus on event agents.

Time-Based Agents

The action of a time-based agent is triggered by a timer. A typical use for a time-based agent is to implement clean-up procedures. An example of a time-based agent is the Oracle *i*FS standard Garbage Collector agent, which can be configured to run one or more times each day. Time-based agents can be used to improve system performance by scheduling tasks that may be CPU-intensive to run outside of peak hours.

Event-Based Agents

A typical use for an event-based agent is to provide a log of changes to a sensitive file. An example of this type of event agent is a custom agent that provides a change log for accounting records, indicating that a specific file was updated at a certain time by a certain person.

Combination Agents

A single agent can be written to respond to both time-related and event-based actions. An example of this type of agent is the Oracle *i*FS Quota agent. The Quota agent is designed to perform a quota check based on "New Document" events, and also to perform periodic timed checks on users who are logged into Oracle *i*FS.

How Do Agents Work?

Agents differ from other programs in that they do not run independently. The ServerManager is responsible for running the agent. The close interaction between the ServerManager and the agent program can be summarized as follows:

- The set of agents is determined by the ServerManager configuration file.
- Agent programs run under the control of the ServerManager. The ServerManager is used to start, stop, suspend, or resume agents. The ServerManager controls three types of agents:
  - Agents that run protocol servers
  - Oracle *i*FS agents
  - Custom agents
- When an agent is started, the agent program registers with the Oracle *i*FS repository, to receive the events the agent will act upon, and with the ServerManager to receive timed events.

The SalaryFileLog Agent at Work

To understand the process of using an agent, consider the example of an agent created to log changes to a sensitive file called CurrentSalary. Assume that:

- A custom Document subclass called CurrentSalary has been defined.
- A SalaryFileLog agent has been developed using the *i*FS agent framework and API. The SalaryFileLog agent class contains the code for the action this agent should take. Specifically, the agent adds a line to a logfile whenever a CurrentSalary object is created, changed, or deleted.
- The SalaryFileLog agent has been registered with ServerManager by adding it to the ServerManager configuration file.

Here is the sequence of events that will occur:

1. Once the agent is started and registered, the agent "goes to sleep," waiting for events from the repository.
2. When the CurrentSalary file is updated, the Oracle *i*FS repository "wakes up" the SalaryFileLog agent by notifying the agent that a registered event has occurred.
3. The SalaryFileLog agent processes the event, adding a line to a log file. When the agent has completed its action, it "goes to sleep" again until the Oracle *i*FS repository notifies it that the next registered event has occurred.

These three steps repeat until the ServerManager stops the agent.

Classes and Methods for an Event Agent

Because agents interact so closely with the ServerManager, it may be useful to look at writing a custom agent in terms of providing the items the ServerManager requires to successfully run the agent. The following table summarizes what the ServerManager requires and lists the Oracle *i*FS API classes or methods used to meet the requirement.

Step	Requirement	Related Class or Method
1.	A Java class that: - Extends IfsAgent - Implements IfsEventHandler	`oracle.ifs.agents.common.IfsAgent` `oracle.ifs.common.IfsEventHandler`
2.	To instantiate the class, the agent needs a constructor. The agent must register to publish status.	`registerDetails()`
3.	A valid Oracle *i*FS connection.	`connectSession()`

4,	A method to register this agent for the object or objects it will act upon:
	- A specific object
	- All instances of a specific class
	Corresponding methods to deregister the agent.
		`LibrarySession.registerEventHandler()`
		`LibrarySession.registerClassEventHandler()`
		`LibrarySession.deregisterEventHandler()`
		`LibrarySession.deregisterClassEventHandler()`

5.	A method to respond to a Start request: - Start the agent and call its methods	`run()`

6.	Methods to handle the agent cycle:
	- Handle requests
	- Process events
	- Wait
		`handleRequests()`
		`processEvents()`
		`waitAgent()`

Step	Requirement	Related Class or Method

7.	Methods to deal with event listening:
	- To enable event listening
	- To disable event listening
		`enableEventListening()`
		`disableEventListening()`

8.	Event handling:
	- A method to receive the event and queue it.
	- A method to process the event.
	- Custom code for processing
		`oracle.ifs.common.IfsEventHandler.`
		`handleEvent()`
		`queueEvent()`
		`processEvent()`
		A custom method, such as `logObjectFolderPath()`

9.	Response to ServerManager requests:
	- Publish detail in response to a Start request
	- Stop request
	- Clean up after the agent runs
	- Suspend request
	- Resume request
		`publishStatusDetail()`
		`handleStopRequest()`
		`postRun()`
		`handleSuspendRequest()`
		`handleStopRequest()`

Writing an Event Agent

Here is the structure of a typical event agent program:

1. Declare the Class
2. Create the Constructor
3. Write the run() Method
4. Handle a Stop Request
5. Handle a Suspend Request
6. Handle a Resume Request
7. Handle Oracle iFS Events

Once the agent has been created, you must configure the ServerManager to instantiate and run the agent. For information about this process, see "Registering an Agent with ServerManager".

The event agent sample code used to illustrate this process is for the ColorAgent, which is registered to receive notice of events on instances of the class MyColorObjects.

Start with Template Code

The simplest way to write an agent is to use Oracle *i*FS sample code as a template. The Oracle *i*FS sample code provides a structure to manage interaction with the ServerManager, allowing you to focus on creating the custom code for your agent's specific task.

The agent template sample code, called `LoggingAgent.java`, is available in the Documentation section of the Oracle *i*FS listing on OTN (Oracle Technology Network).

To aid in planning your custom agent, the following table specifies, for each section, whether the section:

- Uses the generic Oracle *i*FS template code.
- Requires custom code.

Section	Use Generic Code As Is	Optionally Override Generic Code	Custom Code Required
1. Declare the Class	Yes		
2. Create a Constructor	Yes		
3. Write the run() Method		Yes	
4. Handle a Stop Request		Yes	
5. Handle a Suspend Request		Yes	
6. Handle a Resume Request		Yes	
7. Handle Oracle *i*FS Event			Yes

Declare the Class

Every event agent must:

- Extend the class `oracle.ifs.agents.common.IfsAgent`.
- Implement the interface `oracle.ifs.common.IfsEventHandler`.

The IfsEventHandler interface defines the methods that must be provided for a custom event agent. Both event agents and combination agents, ones that react to both class-based and time-based events, must include both the class and the interface in the class declaration.

Note: The agent name must be one word; no embedded spaces are allowed.

The Class Declaration

This sample provides all of the required code for this section.

Sample Code: The Class Declaration

```
public class ColorAgent extends IfsAgent implements IfsEventHandler
```

Create the Constructor

Every agent must implement the standard constructor for an agent.

The Constructor Method

This sample provides all of the required code for this section. Simply replace the agent name ColorAgent with the name of your custom agent.

Sample Code: The Constructor Method

```
public ColorAgent(String name, String[] args, String sectionName,
        ServerManager manager) throws IfsException
{
   super(name, args, sectionName, manager);
}
```

Write the run() Method

The following code sample demonstrates registering the agent with the ServerManager and activating event listening for this agent.

The run() method is closely associated with several other methods, so those code samples are also shown here:

- The run() method calls enableEventListening() to register for specific events.
- When the ServerManager breaks out of the run() loop, postRun() is called.
- The postRun() method calls disableEventListening() to deregister the agent.

The run() Method

This code sample performs the following tasks:

- Perform setup tasks:
  - Write a message to the log file for this agent.
  - Perform any agent-specific initialization.
  - Ensure that a connection exists.

    - Enable event listening.
    - Call the ServerManager to publish that this agent is available.
- Start the run() loop by checking whether the agent has been instructed to stop. (If so, break out at this point.) If the agent is not stopped:
    - Check for any ServerManager requests (Start, Resume, Suspend, Stop).
    - Write a message to the log file when agent status changes.
    - Check for any events to process (a call to the processEvents() method).
    - Once an event is processed, return to a Wait state.

To provide a concise example, this sample does not include exception handling for the run() method. Because it is particularly serious if errors occur in the run() method, the full code example includes comprehensive exception processing. To view the exception handling section, see the full text in the "Event Agent (Complete Code Example)", at the end of this chapter.

Adding Custom Code The following sample provides the minimum required code for this section.You can add additional custom code in this section to perform any agent-specific initialization required.

Sample Code: The run() Method

```
public void run()
{
    try
    {
        log("Start request");

        // perform any agent-specific startup initialization
        // (none currently)

        // ensure a connection
        connectSession();

        // enable event listening
        enableEventListening();

        // declare ourselves up
        publishStatusDetail();

        while (true)
        {
            if (!isAlive())
            {
                log("Exiting handle loop");
                break;
            }

            try
            {
                // handle any status changes first
                handleRequests();

                // process events
                processEvents();

                // wait for something to do
                waitAgent();
            }
        }
    }
    catch (Exception e)
    {
    }
}
```

The postRun() Method

The following code sample is automatically executed by the ServerManager when the agent breaks out of or returns from the run() method. It provides a place to add code for any required clean-up.

In this code sample, the following tasks are performed:

- Any special shutdown tasks required by your custom agent.
- Disable EventListening.
- Disconnect the session.

Sample Code: The postRun() Method

```
public void postRun()
{
    super.postRun();

    log("postRun");

    try
    {
        // Perform any special shutdown tasks (none currently)
        // Disable event listening
        disableEventListening();

        // Disconnect our session.
        disconnectSession();
    }
    catch (Exception e)
    {
    }
}
```

The enableEventListening() Method

The following code sample registers this agent for events on a specific class by calling the registerClassEventHandler() method. In this code sample, the following tasks are performed:

- The registerClassEventHandler() method is called for the Oracle *i*FS class of interest. This section of the code allows the agent to receive events for all *new* instances of the registered class.
- A selector object is created to hold the results of a search for existing objects of the specified class. This section of the code allows the agent to receive events for all *existing* instances of the registered class.

Sample Code: The enableEventListening() Method

```
public void enableEventListening() throws IfsException
{
    try
    {
        Library Session sess = getSession();
        // Register for events on all new MyColorObjects.
        Collection c = sess.getClassObjectCollection();
        ClassObject myColorClass =
            (ClassObject)c.getItems(MyColorObject.CLASS_NAME);
        sess.registerClassEventHandler(myColorClass, true, this);

        // Also select all existing MyColorObject objects so that we
        // get events on them also.
        Selector selector = new Selector(sess);
        selector.setSearchClassname(MyColorObject.CLASS_NAME);
        selector.setSearchSelection(null);
        selector.getItems();
    }
    catch (Exception e)
    {
    }
}
```

The disableEventListening() Method

The following code performs the corresponding deregistration for this agent by calling the deregisterClassEventHandler() method.

Sample Code: The disableEventListening() Method

```
public void disableEventListening()
{
    try
    {
        LibrarySession sess = getSession();
        // Deregister for events on all MyColorObjects.
        Collection c = sess.getClassObjectCollection();
        ClassObject myColorClass =
            (ClassObject)c.getItems(MyColorObject.CLASS_NAME);
        sess.deregisterClassEventHandler(myColorClass, true, this);
    }
    catch (Exception e)
    {
    }
}
```

Handle a Stop Request

Every agent must handle the three possible requests from the ServerManager:

- Stop
- Suspend
- Resume

The following code sample demonstrates handling a Stop request from the ServerManager.

The handleStopRequest() Method

This section of the agent program makes the call to the IfsAgent.handleStopRequest() method, which logs an indication of the agent's status change.

Adding Custom Code The following sample provides the minimum required code for this section.You can add additional custom code in this section to perform custom tasks as part of a Stop request.

Sample Code: The handleStopRequest() Method

```
protected void handleStopRequest() throws IfsException
{
    // the super sets our status to "stopping"
    super.handleStopRequest();
    log("Stop request");
}
```

Handle a Suspend Request

The following code sample demonstrates handling a Suspend request from the ServerManager.

The handleSuspendRequest() Method

This section of the agent program makes the call to the IfsAgent.handleSuspendRequest() method. In this code sample, the following tasks are performed:

- Log an indication of the status change.
- Disable EventListening.

Adding Custom Code The following sample provides the minimum required code for this section.You can add additional custom code in this section to perform custom tasks as part of a Suspend request.

Sample Code: The handleSuspendRequest() Method

```
protected void handleSuspendRequest() throws IfsException
{
    // the super sets our status to "suspended"
    super.handleSuspendRequest();

    log("Suspend request");

    // disable our event listening & timer
    disableEventListening();
}
```

Handle a Resume Request

The following code sample demonstrates handling a Resume request from the ServerManager.

The handleResumeRequest() Method

This section of the agent program responds to a Resume request from the ServerManager. A Resume request is the opposite of a Suspend request. This section makes the call to the IfsAgent.handleResumeRequest() method.

In this code sample, the following tasks are performed:

- Log an indication of the status change.
- Re-enable event listening.

Adding Custom Code The following sample provides the minimum required code for this section.You can add additional custom code in this section to perform custom tasks as part of a Resume request.

Sample Code: The handleResumeRequest() Method

```
protected void handleResumeRequest() throws IfsException
{
    // the super sets our status to "started"
    super.handleResumeRequest();

    log("Resume request");

    // re-enable event listening
    enableEventListening();
}
```

Handle Oracle *i*FS Events

The Handle Oracle *i*FS Events section of the agent program consists of two methods:

- The handleEvent() method queues the events for processing by the processEvent() method.
- The processEvent() method holds the key block of code in the agent program. Use this method to hold the custom code you write to implement the agent's action on an event.

The handleEvent() Method

If there is any category of events that you do not want to process, you can use this method as a filter to exclude those events. In this example, we do not want to process any MyColorObjects that have been deleted from Oracle *i*FS.

Sample Code: The handleEvent() Method

```
public void handleEvent(IfsEvent event)
{
    // do not queue any FREE events, but queue all others
    if (event.getEventType() != IfsEvent.EVENTTYPE_FREE)
    {
        queueEvent(event);
        notifyAgent();
    }
}
```

The processEvent() Method

The processEvent() method calls the logObjectFolderPath() method to do the actual work of the agent.

Sample Code: The processEvent() Method

```
public void processEvent(IfsEvent event) throws IfsException
{
    // log the object with its folder path
    logObjectFolderPath(event);
}
```

The logObjectFolderPath() Method

The processEvent() method calls the logObjectFolderPath() method to do the actual work of the agent, which involves obtaining the pieces of information that make the message that will be printed to the log file. Because this sample is trivial, we could have included this code in the processEvent() method. We have shown it as a separate method because a custom agent might well require several methods to perform its work.

Because this is sample code, the agent performs the minimal action of printing out that the agent received the event for the object. However, this is the location where you should add the code for whatever action you want the agent to take on this object, such as sending a notification.

Sample Code: The logObjectFolderPath() Method

```
public void logObjectFolderPath(IfsEvent event) throws IfsException
{
    try
    {
        Long objectId = event.getId();
        int eventType = event.getEventType();
        MyColorObject colorObject =
            (MyColorObject)getSession().getPublicObject(objectId);

        // Get any folder path to this object, and log it
        String objectPath = colorObject.getAnyFolderPath();
        log("Received Event Type " + eventType
            + " on object " + objectPath);
    }
    catch (Exception e)
    {
    }
}
```

Registering an Agent with ServerManager

All agents run within an instance of the Oracle *i*FS ServerManager. Although the system administrator will configure definition files for the primary ServerManager, you as a developer will need to create your own definition (.def) files to test your custom agents.

During testing, you can run your agent in a standalone mode, using only your specific .def file. During system integration and for production, you will want to have the system administrator add your custom agent configuration parameters to the primary Oracle *i*FS ServerManager agent definition file.

The following steps present a high-level view of the registration process for running the agent in standalone mode for testing, assuming that you have already created and compiled the custom agent class:

1. Log in to the Solaris environment.
2. Create a definition file for your custom agent.

   See "Sample Code: Agent Definition File" for a sample definition file, `CustomServerManager.def`.
3. Use the following command to run a standalone instance of ServerManager for your custom agent:

```
$ ifssvrmgr CustomServerManager.def
```

In this example, the `$` represents the Solaris prompt. At the Solaris prompt, enter the command `ifssvrmgr` followed by the name of your .def file. Substitute the name of your file for *`CustomServerManager.def`* in the example above.

Agent Definition File

The CustomServerManager.def file registers an agent called "ColorAgent" with the ServerManager. This sample file registers just one custom agent; it could also be used to register multiple agents. Store this file on the Oracle *i*FS server machine wherever .def files for other agents are stored (usually `$ORACLE_HOME/ifs/settings`.)

Sample Code: Agent Definition File

```
; CustomServerManager.def
; This file includes the Color Agent.
;
; ServerManager Configuration Information
ManagerName = CustomServerManager
Interactive = false
Outputfile = /myDirectory/CustomServerManager.log
;
; The name of the agent.
Agents += ColorAgent
;
; Description of the agent.
;
[ColorAgent]
Name = ColorAgent
Class = oracle.ifs.agents.examples.ColorAgent
Start = true
```

The following table describes the parameters in the agent definition file:

Parameter	Description
`ManagerName`	Unique name of this instance of ServerManager.
`Interactive`	Whether or not this instance of ServerManager runs in interactive mode.

Parameter	Description
`Outputfile`	Fully qualified pathname for the log file for this instance of ServerManager.
`Name`	Specifies the name of the agent. Must be one word.
`Class`	Specifies the package hierarchy for the agent `.class` file. Verify that there is an entry in the `CLASSPATH` environment variable that the system can use to locate the `.class` file for the custom agent.
`Start`	Sets whether to start the agent automatically when this instance of ServerManager is started. If set to `false`, the agent must be started manually.

Testing the Agent

Once the agent is complete and has been registered, your testing scenario will look something like this:

- Start ServerManager.
- Start the custom agent. (In ServerManager, list the agents to verify that your agent has been started.)
- Perform a task that your agent should detect.
- Verify that your agent carries out the prescribed task, such as printing a line to the log file.

Event Agent (Complete Code Example)

The following agent, ColorAgent, is registered to receive notice of events on objects of the class MyColorObjects.

Sample Code: Event Agent

```
/* --ColorAgent.java-- */
/package oracle.ifs.agents.examples;

import oracle.ifs.common.Collection;
import oracle.ifs.common.IfsEvent;
import oracle.ifs.common.IfsEventHandler;
import oracle.ifs.common.IfsException;
import oracle.ifs.common.ParameterTable;
import oracle.ifs.beans.ClassObject;
import oracle.ifs.beans.LibrarySession;
import oracle.ifs.beans.PublicObject;
import oracle.ifs.beans.Selector;

//This is the custom class this agent deals with.
import ifs.demo.colors.type.MyColorObject;

import oracle.ifs.agents.common.IfsAgent;
import oracle.ifs.agents.manager.ServerManager;

/**
 * A ColorAgent logs messages according to events that are raised
 * on the class, MyColorObject.
 */
public class ColorAgent extends IfsAgent implements IfsEventHandler
{
    /**
     * Constructs a ColorAgent.
     */
    public ColorAgent(String name, String[] args, String sectionName,
        ServerManager manager) throws IfsException
    {
        super(name, args, sectionName, manager);
    }

    /**
     * Runs this ColorAgent.
     */
    public void run()
    {
        try
        {
            log("Start request");

            // perform any agent-specific startup initialization
            // (none currently)

            // ensure a connection
            connectSession();

            // enable event listening
            enableEventListening();

            // declare ourselves up
            publishStatusDetail();

            while (true)
            {
                if (!isAlive())
                {
                    log("Exiting handle loop");
                    break;
                }

                try
                {
                    // handle any status changes first
                    handleRequests();

                    // process events
                    processEvents();

                    // wait for something to do
                    waitAgent();
                }
                catch (IfsException e)
                {
                    log("IfsException" + " in handle loop; continuing:");
                    log(e.toLocalizedString(getSession()));
                }
                catch (Exception e)
                {
                    log("Exception" + " in handle loop; continuing:");
                    log(e.getMessage());
                }
            }
        }
        catch (IfsException e)
        {
            // exception that takes us out of the run loop;
            // this will cause a stop.
            log("IfsException" + " in run(): ");
            log(e.toLocalizedString(getSession()));
            printStackTrace(e);
        }
        catch (Exception e)
        {
              // exception that takes us out of the run loop;
              // this will cause a stop.
```

```
            log("Exception" + " in run(): ");
            log(e.getMessage());
            printStackTrace(e);
        }
    }

    /**
     * Handle the Stop request.  Subclasses can override this to
     * perform custom tasks as part of a Stop request.
     */
    protected void handleStopRequest() throws IfsException
    {
        // the super sets our status to "stopping"
        super.handleStopRequest();

        log("Stop request");
    }

    /**
     * Handle the Suspend request.  Subclasses can override this to
     * perform custom tasks as part of a Suspend request.
     */
    protected void handleSuspendRequest() throws IfsException
    {
        // The super sets our status to "suspended"
        super.handleSuspendRequest();

        log("Suspend request");

        // Disable our event listening
        disableEventListening();
    }

    /**
     * Handle the Resume request.  Subclasses can override this to
     * perform custom tasks as part of a Resume request.
     */
    protected void handleResumeRequest() throws IfsException
    {
        // the super sets our status to "started"
        super.handleResumeRequest();

        log("Resume request");

        // re-enable event listening
        enableEventListening();
      }

     /**
      * Performs post-run tasks for this ColorAgent.
      */
    public void postRun()
    {
        super.postRun();

        log("postRun");

        try
        {
            // perform any special shutdown tasks (none currently)

            // disable event listening
            disableEventListening();

            // Disconnect our session.
            disconnectSession();
        }
        catch (Exception e)
        {
            log("Exception" + " in postRun:");
            log(e.getMessage());
        }
     }
    /**
     * Enable listening for MyColorObject events
     */
    public void enableEventListening() throws IfsException
    {
        try
        {
            // Register for events on all MyColorObjects
            Collection c = getSession().getClassObjectCollection();
            ClassObject myColorClass =
                (ClassObject)c.getItems(MyColorObject.CLASS_NAME);
            getSession().registerClassEventHandler(myColorClass, true, this);

            // also select all MyColorObject objects so that we get events
            // on any of them
            Selector selector = new Selector(getSession());
            selector.setSearchClassname(MyColorObject.CLASS_NAME);
            selector.setSearchSelection(null);
            selector.getItems();
        }
        catch (IfsException e)
        {
            log("IfsException" + " enabling Event Listening; re-throwing:");
            log(e.toLocalizedString(getSession()));
            printStackTrace(e);
            throw e;
        }
        catch (Exception e)
        {
            log("Exception" + " enabling Event Listening; re-throwing:");
            log(e.getMessage());
            printStackTrace(e);

            // throw "agent unable to enable event listening"
            throw new IfsException(46002, e);
        }
    }

    /**
     * Disable listening for MyColorObject events
     */
    public void disableEventListening()
    {
        LibrarySession session = getSession();
        try
        {
            // Deregister for events on all MyColorObjects.
            Collection c = getSession().getClassObjectCollection();
            ClassObject myColorClass =
                (ClassObject)c.getItems(MyColorObject.CLASS_NAME);
            session.deregisterClassEventHandler(myColorClass, true, this);
        }
        catch (Exception e)
        {
            log("Exception" + " disabling Event Listening:");
            log(e.getMessage());
        }
    }

    /**
     * Handles events on MyColorObjects.  This queues the events for
     * processing by the main agent thread.
```

```
 */
public void handleEvent(IfsEvent event)
{
    // do not queue any FREE events, but queue all others
    if (event.getEventType() != IfsEvent.EVENTTYPE_FREE)
    {
        queueEvent(event);
        notifyAgent();
    }
}

/**
 * Process the de-queued event.  The processing is simply to log
 * a message with the object's name and folder path.
 *
 *  @param event            The event to log
 *
 */
public void processEvent(IfsEvent event) throws IfsException
{
    // log the object with its folder path
    logObjectFolderPath(event);
}
/**
 * log information about an object received as an event.
 *
 *  @param event            the event to log
 *
 */
public void logObjectFolderPath(IfsEvent event) throws IfsException
{
    try
    {
        Long objectId = event.getId();
        int eventType = event.getEventType();
        MyColorObject colorObject =
            (MyColorObject)getSession().getPublicObject(objectId);

        // get any folder path to this object, and log it
        String objectPath = colorObject.getAnyFolderPath();
        log("Recieved Event Type " + eventType
            + " on object " + objectPath);
    }
    catch (IfsException e)
    {
            // exception getting the information about the obejct
            // referred in the event
            log("IfsException" + " in processEvent(): ");
            log(e.toLocalizedString(getSession()));
        }
        catch (Exception e)
        {
            // exception getting the information about the obejct
            // referred in the event
            log("Exception" + " in processEvent(): ");
            log(e.getMessage());
        }
    }
}
//EOF
```

9

Using Overrides

This chapter covers the following topics:

What Is an Override?

Overrides belong to the category of Oracle Internet File System customization, which includes:

- Creating custom classes (types)
- Parsers and renderers
- Agents
- Overrides

Of these four types of customization, overrides present a significant leap in complexity. Except for renderers, the other types of customization take place on the client, or "bean-side." These more common types of customization use the extensive classes found in the `oracle.ifs.beans` package. With overrides, we tackle "server-side" processing, using a more limited published API.

An override is a method that allows an application program to intervene in a predefined way with the standard repository operations:

- Insert
- Update
- Free

Note: For the 1.1 release of Oracle *i*FS, overrides are applicable only to Insert, Update, and Free operations. No overrides are available for other database operations, such as copy, add item to folder, or remove item from folder. This functionality is planned for future releases of the product.

How Pre- Overrides Work

The Pre- overrides work by allowing you to interrupt the standard flow of processing on the server side at certain predefined points. At these points, you can specify custom processing to occur before one or more of the standard database operations takes place. For example, your custom validation routine might be incorporated into both PreInsert and PreUpdate overrides.

Using Pre- Overrides

The behavior of the actual Insert, Update, and Free operations is complex, and to override the actual operations risks producing unintended and unwanted results.

However, in almost every circumstance, application requirements can be met quite safely by using Pre- methods to interact with the repository immediately before the specified operation. Because the Pre- overrides are the most frequently used, the balance of this chapter focuses on Pre- overrides.

The following table presents an example of how each override method might be used.

Operation	Override Method	Usage Example
Insert	extendedPreInsert()	To add a certain attribute with a specified value to every new instance of a custom document class.
Update	extendedPreUpdate()	Same as for Insert, to extend this processing to Update operations. Or to provide special validation on Update.
Free	extendedPreFree()	To write a copy to a Deletions log file.

Before You Begin Working with Overrides

Because working with overrides is a complex task, we recommend that you postpone tackling overrides until you have had considerable experience with Oracle *i*FS and its Java API. Specifically, we suggest that your background should include knowledge and hands-on experience in the following areas:

- Sufficient familiarity with the bean-side API to know how Oracle *i*FS objects work:
  - How objects are created.
  - How object attributes are set.
  - How related objects, such as Documents and Folders, work together.
- Sufficient familiarity with attribute values and the definition classes to know how they work together:
  - How to use the AttributeValue class, the Oracle *i*FS data structure that holds attribute values in a definition object.
  - How to set and get bean-side attribute values.
  - How to set and get server-side attribute values.
  - How to work with sets of attribute values.
  - How to use basic datatypes to define attribute values.
  - How to use value defaults.
  - How to use value domains.
- Familiarity with the Oracle *i*FS transaction model.

Providing this range of information is beyond the scope of this Guide. To gain the experience required to tackle overrides, we suggest beginning with several less-complex Oracle *i*FS applications. This hands-on experience, combined with attendance at Oracle *i*FS training, will provide the needed familiarity with the Oracle *i*FS objects and the way they work together.

Review of Attributes

If you have created Oracle *i*FS objects using the classes of the Oracle *i*FS API, you will recall the two-phase method of object creation:

1. Create a Definition object.
2. Pass the Definition object to the method that actually creates the object.

For example, for a document, you would first create a DocumentDefinition object, then pass that object as an argument to the LibrarySession.createPublicObject() method. For more information, see "Creating a New Document" in Chapter 3, "Working with Documents".

Attributes and Server-Side Classes

In the Oracle *i*FS Java class hierarchy each Oracle *i*FS object has two representations:

- The bean-side representation, known by the object name, such as Document.
- The server-side representation, known by the object name preceded with "S_", such as S_Document.

When you work with overrides, you are working with objects in the Oracle *i*FS repository. In this context, you use the S_ classes, such as S_Document, rather than the familiar bean-side classes, such as Document.

Attributes and Special Options

The function of the Definition object is to specify the object's attributes. When you create a DocumentDefinition object, you set two types of information:

- Document attributes:
  - Document attributes, such as Name and Description, are the metadata about the document that can be easily searched.
  - To set an attribute, use the generic setAttribute() method defined in the LibraryObjectDefinition class.
- Special options:
  - Special options, such as AddToFolderOption, specify detailed information about the document but do not provide attribute values.

- To set special options, use the specific method provided in the Definition class, such as the oracle.ifs.beans.PublicObjectDefinition. setAddToFolderOption().
- Although special options are set on the bean-side, they are handled on the server side, using the S_LibraryObjectDefinition object.

User-set Attributes and Derived Attributes

Attributes can also be divided into two categories based on their origin:

- User-set attributes

  User-set attributes are attributes that are set by and can be updated by the user, such as the object Name and Description.

  You can set all the user-set attributes using the bean-side Definition object.

- Derived attributes

  Derived attributes are attributes that are set by and can only be updated by the system, such as Creator. These attributes are derived from a source other than the user. For example, Creator is derived from the current logon ID.

  To handle the derived attributes, you must use the server-side class, S_DocumentDefinition.

Override Methods

The override methods provide pre-processing for the standard database operations:

- Insert
- Update
- Free (for override purposes, use the methods related to Free rather than Delete)

Because overrides by their nature take place on the server-side, all the override methods are located in the S_LibraryObject class and its subclasses. Just as oracle.ifs.*beans*.LibraryObject provides a bean-side Java representation for all the objects that end users manipulate directly (such as documents and folders), oracle.ifs.*server*.S_LibraryObject provides the server-side Java representation for these same objects.

Operation	Method	Purpose
Insert		
	extendedPreInsert	Performs designated operations before inserting an *i*FS object into the database. For example, used to modify any attributes after the Definition object has been created but before the Insert takes place.
Update		
	extendedPreUpdate	Performs designated operations before updating an *i*FS object in the database.
Free		
	extendedPreFree	Performs designated operations before freeing a database object. Note that this method is overridden by classes that need to perform operations before successfully deleting the rows for the freed instance.

Writing an Override

To plan an override, first decide which operation or operations you want to override (Insert, Update, or Free).

To write an override, follow these steps:

1. Declare the Server-side Class.
2. Create the Constructor.
3. Implement the Override Method.

Declare the Server-side Class

Assume you have already created a bean-side custom class called PurchaseOrder that extends the Document class. Now you decide you want to add some special validation checks via a PreInsert override.

First you create a server-side Java class, S_PurchaseOrder, to represent your custom class in the server. S_PurchaseOrder extends S_TieDocument. Tie classes allow you to "tie" into the class hierarchy at any level, letting you customize without changing the way the class hierarchy is structured. For more information, see "Tie Classes" in Chapter 2, "API Overview".

Put this class into a new custom package, such as `MyCompany.MyApp.server.` (Do not add this class to the `oracle.ifs.server` package).

Sample Code: Declare the Class

```
public class S_PurchaseOrder extends S_TieDocument
```

Create the Constructor

Every override class must implement two constructors:

- One used for an object that currently exists in the database.
- One used for an object that has not yet been created in the database.

Sample Code: Constructor

```
public S_PurchaseOrder(S_LibrarySession session, S_LibraryObjectData data)
      throws IfsException
```

Parameters	Datatype	Description
`session`	S_LibrarySession	Current LibrarySession.
`data`	S_LibraryObjectData	Data component

Sample Code: Constructor

```
public S_PurchaseOrder(S_LibrarySession session, java.lang.Long classID)
      throws IfsException
```

Parameters	Datatype	Description
`session`	S_LibrarySession	Current LibrarySession.
`classID`	Long	Class ID for the object that is in the process of being created.

Implement the Override Method

The S_PublicObject class provides Pre- methods for the standard database operations. For a complete list, see "Override Methods".

The following table describes the parameters of the Override methods.

Parameter	Datatype	Description
`opState`	OperationState	Used by the system to track the current state of operations.
`def`	S_LibraryObjectDefinition	Current object definition to be updated with system attributes.

Sample Code: Implement the Override Method

```
public void extendedPreInsert(OperationState opState,
          S_LibraryObjectDefinition def) throws IfsException

{
     super.extendedPreInsert(opState, def);
     //Add your validation code here.
}
```

The first call after the method begins should be to "super." This call implements processing from the superclass of this object. In this case, it allows both the S_Document class and the S_PublicObject class to perform processing.

After that, add any specific validation code that your application requires. For example, in the custom bean-side class for Purchase Order, you may have added a setApprover() method. Before you insert the PurchaseOrder object into the database, you may want to check the value of that Approver, and perform one of the following actions:

- Validate that the Approver is correct.
- Add a default Approver if the current value is null.
- Change one Approver name to another based on some custom logic.

Sample Code: A PreInsert Override

The following sample code provides a brief example of using a PreInsert override to add server-side validation. Note the following three lines shown in bold, where placeholders are used that must be replaced with appropriate code for your application:

- mypackage
- <valid>
- <your custom error code>

```
// S_PurchaseOrder.java

package mypackage;


import oracle.ifs.common.AttributeValue;
import oracle.ifs.common.IfsException;

import oracle.ifs.server.S_DirectoryUser;
import oracle.ifs.server.S_LibraryObjectData;
import oracle.ifs.server.S_LibraryObjectDefinition;
import oracle.ifs.server.S_LibrarySession;
import oracle.ifs.server.OperationState;
import oracle.ifs.server.S_TieDocument;


// server side PurchaseOrder class example
public class S_PurchaseOrder extends S_TieDocument
{
     // constructors
     public S_PurchaseOrder(S_LibrarySession session, S_LibraryObjectData data)
         throws IfsException
     {
         super(session, data);
     }
     public S_PurchaseOrder(S_LibrarySession session, Long classId)
         throws IfsException
     {
         super(session, classId);
     }
```

```
    // override Pre Insert Operation
    public void extendedPreInsert(S_LibraryObjectDefinition def,
                                  OperationState opState) throws IfsException
    {
        // ALWAYS call super
        super.extendedPreInsert(opState, def);

        // get our session
        S_LibrarySession session = getSession();

        // get the approver attribute so we can check it
        AttributeValue av1 = def.getAttribute("APPROVER");
        S_DirectoryUser approver = (av1 == null) ? null:
            (S_DirectoryUser) av1.getDirectoryObject(session);

        //validate the approver
        if (! <valid> )
        {
             // this will rollback the operation
             throw new IfsException( <your custom error code>, this );
        }

        // otherwise, set the APPROVED bit to true.
        // the APPROVED bit is not settable or updateable
        // by users, but the server can set it thusly
        AttributeValue av2 = AttributeValue.newAttributeValue(true);
        def.setSystemSetAttribute("APPROVED", av2);
   }
}
```

10 Sending E-mail Programmatically

This chapter covers the following topics:

What Is Sending E-mail Programmatically?

We are all familiar with using standard e-mail clients to create and send e-mail messages. However, e-mail can also be created and sent by an application program. This type of e-mail is called programmatic e-mail.

When a user inserts, updates, or deletes a file, your custom application can perform the following tasks:

- Generate an e-mail message notifying one or more other users of the change.
- Send the message using the Oracle Internet File System programmatic e-mail capability.

The message is staged in the Oracle *i*FS Outbox to await delivery.

Programmatic e-mail takes place in a transactional environment. Applications that require transactional capability include multiple activities, one of which is an e-mail notification. For the application to complete the task, the application must perform some action and then send a notification of that action. The action and the notification must be so tightly linked that you want *both* to be rolled back if the message is not accepted by the Outbox. Here are the stages of the process:

- Perform the action.
- Create the message using the Oracle *i*FS e-mail API.
- Insert the message in the Outbox.

An example of this type of application would be creating a new customer account, then sending a message to the customer saying, "Here is your new account number and password." If the message cannot be sent, the account and password are rolled back. The Oracle *i*FS framework for programmatic e-mail provides a mechanism for Oracle *i*FS applications to send such e-mail messages.

Oracle *i*FS Infrastructure for Programmatic E-mail

The Oracle *i*FS infrastructure for programmatic e-mail includes the Outbox folder and the IfsMessage class.

The Outbox folder is the location where messages are staged to await delivery. A system-wide Oracle *i*FS Outbox is created during the installation process. An Outbox agent delivers mail present in the Outbox on an event-driven basis.

The Oracle *i*FS API class, IfsMessage, provides methods to create and send e-mail programmatically. All e-mail consists of three parts:

- Header information, such as To, From, and Subject
- Body (optional)
- Attachments (optional)

The methods of the IfsMessage class allow you to set appropriate values for each of these parts.

Programmatic E-mail Scenario

One common use of programmatic e-mail is to use an application program to respond automatically to customer actions. For example, in an e-commerce application:

- A customer logs onto your company's e-commerce web site.
- The customer submits an order.
- The company wants to send an automatic e-mail confirmation of the order.

To create this automatic e-mail confirmation, you use information from multiple sources. Some information comes from the customer, such as name and e-mail address. Other information comes from your company, such as text for the confirmation message and any attachments, such as an invoice.

The application program must perform three tasks:

- Gather the information from appropriate sources.
- Using the information, create a message object.
- Send the e-mail message to the customer.

Writing an Application to Send E-mail Programmatically

To write a programmatic e-mail application, follow these steps:

1. Create an IfsMessage Object
2. Construct the Message Header
3. Construct the Message Body
4. Send the Message

For more information about e-mail methods, see the Javadoc for the `oracle.ifs.adk.mail.IfsMessage` class.

Option for Sending Short Messages

Note that if you want to send a very short mesage, you can use the convenience method, oracle.ifs.adk.mail.IfsMessage.sendMessage(). To use sendMessage(), supply the following parameters, as appropriate:

Parameter	Datatype
Session	LibrarySession
To	String[]
CC (Carbon Copy)	String[]
BCC (Blind Carbon Copy)	String[]
Subject	String
Body	Reader

Sample Code: Send a Short Message

```
// Use the convenience method to send a short message.
     mail.sendMessage(currentSession,"Recipient", "CcRecipient", "BccRecipient",
           "Subject:Sending Short Messages","This is a one-sentence message.");
```

Create an IfsMessage Object

Creating a database connection is the first task of every Oracle *i*FS application, and requires both LibraryService and LibrarySession. For a complete sample of the standard technique for obtaining the current Library Session, see "Sample Code: Sending E-Mail Programmatically".

In this case, LibrarySession is the single parameter passed to the IfsMessage constructor.

For more information about LibraryService and LibrarySession, see Chapter 2, "API Overview":

- "The LibraryService Class"
- "The LibrarySession Class"

Sample Code: Create an IfsMessage Object

Before you create the message object, check to be sure the current user has Admin privileges.

Note: Only users with Admin Mode enabled can send e-mail programmatically.

```
// Check that user is in Admin Mode.
if (user.isAdminEnabled())
        session.setAdministrationMode(true);

// Use the IfsMessage constructor to create a message object.
     IfsMessage msg = new IfsMessage(session);
```

Construct the Message Header

To construct the message header, supply one or more of the following pieces of information:

Item	Datatype	Required/Optional	Setter
To	String	See Note.	`setToHeader()`
CC (Carbon Copy)	String	See Note.	`setCcHeader()`
BCC (Blind Carbon Copy)	String	See Note.	`setBccHeader()`
From	String or DirectoryUser	Automatically inserted; optional.	`setFromHeader()`
Sender	String or DirectoryUser	Automatically inserted; optional.	`setSenderHeader()`
ReplyTo	String	Optional.	`setReplyToHeader()`
In Reply To	Message	Optional.	`setInReplyToHeader()`
Subject	String	Optional.	`setSubject()`

Using Multiple Values for Header Items

Multiple values are allowed for each of the following header items:

- To
- CC
- BCC
- From
- Reply To

For example, to send the same message to multiple destinations, you could set three To headers and six CC headers. Any combination of the header items is acceptable, as long as at least one recipient is indicated. In other words, each message must have either a To, CC, or BCC specified, so the message can be delivered.

In some cases, one person will send mail on behalf of another person. In that case, you can specify a From value (the author of the message) that varies from the Sender value (the person who sends the message). If setFromHeader() is omitted, the From value will be set to the current user by default. If setSenderHeader() is provided, but not setFromHeader(), the value provided for Sender will be used for From and Sender will be omitted.

For details of the setter methods for the message header, see the Javadoc for `oracle.ifs.adk.mail.IfsMessage.` All of the setter methods are overloaded to provide one method that accepts a single String value and another that accepts a String array.

Sample Code: Construct the Message Header

```
msg.setFromHeader("tuser99@us.oracle.com");
msg.setSenderHeader(user);  //Refers to a DirectoryUser
msg.setToHeader("guest@us.oracle.com");
msg.setBccHeader("tuser51@us.oracle.com");
msg.setSubject("iFS email API: Mail with multi-part body");
```

Construct the Message Body

To construct the message body, you can use any of the variants of the setBody() method, and, optionally, of the attach() methods.

Methods	Datatype	Required/Optional	Item
`setBody(String body)`	String	Optional.	Body
`setAlternativeBodies (IfsMessage.MimeBodyPart[])`	MimeBodyPart[]	Optional.	Alternative bodies
`setMixedBodies (IfsMessage.MimeBodyPart[])`	MimeBodyPart[]	Optional.	Mixed bodies
`setParallelBodies (IfsMessage.MimeBodyPart[])`	MimeBodyPart[]	Optional.	Parallel bodies
`attach(String path)`	String	Optional. Use to indicate path to document file.	Attachment
`attach(String[] path)`	String[]	Optional. Use to indicate an array of paths to document files.	Attachment
`attach(Document doc)`	Document	Optional. Attach one document.	Attachment
`attach(Document[] doc)`	Document[]	Optional. Attach an array of documents.	Attachment

Using the setBody() methods

The setBody() method is overloaded to accept several combinations of arguments, and includes 11 variants. For more information on methods used to construct the message body, see the Javadoc for `oracle.ifs.adk.mail.IfsMessage.`

Here are some basic guidelines:

- Use setBody() for a message with no attachments.
- If the message has an attachment, you must use one of the three set*Xxx*Bodies() methods:
  - setMixedBodies()
  - setAlternativeBodies()
  - setParallelBodies()

The setMixedBodies() Method

The setMixedBodies() method will meet most application needs for sending any message with an attachment. The multipart/mixed content type is the most general MIME content type, and can be used for any multipart message or any message with attachments. The client is free to choose the method to display the message. Generally, the different body parts are treated as different parts of the message; that is, they are treated the same as if they were attachments.

When you pass only a single parameter to setMixedBodies(), rather than an array, setMixedBodies has the same effect as the setBody() method.

The setAlternativeBodies() Method

Use setAlternativeBodies() to specify alternative formats for use by specific browsers. A multipart/alternative body is a body of multiple body parts of the same content but in different formats. A compatible client should choose the most appropriate format to display the message to the user.

The content type of the message is always multipart/alterative if the message does not have attachments. Otherwise, the message content type will be multipart/mixed with a nested multipart/alternative message.

When you pass only a single parameter to setAlternativeBodies(), rather than an array, setAlternativeBodies() has the same effect as the setBody() method.

The setParallelBodies() Method

Use setParallelBodies() when all attachments are of a format that can be displayed following the message, rather than included as a separate file. A multipart/parallel body is a message body with multiple body parts that should be viewed in parallel using a compatible client.

The content type of the message is always multipart/parallel if the message does not have attachments. Otherwise, the message content type will be multipart/mixed with a nested multipart/parallel message.

When you pass only a single parameter to setParallelBodies(), rather than an array, setParallelBodies() has the same effect as the setBody() method.

Sample Code: Construct a Simple Message Body

```
msg.setBody("Hello there! This is a simple string body.");
```

Sample Code: Construct a Multipart Message Body

```
//Construct a multipart message body using setAlternativeBodies().
MimeBodyPart[] bodies = new MimeBodyPart[2];
bodies[0] = new MimeBodyPart("Hello, this is plain text.",
                               "text/plain", null);
bodies[1] = new MimeBodyPart("<html><body>Hello, this is HTML text " +
                               ".</body></html>", "text/html", null);
msg.setAlternativeBodies(bodies);
```

In this example, note the class MimeBodyPart. MimeBodyPart is a class that represents a single body part of a multipart MIME message. This class provides a means to group the essential information of a message body part into a single class for ease of storage and retrieval. MimeBodyPart is an inner class of the IfsMessage class; many methods of IfsMessage accept MimeBodyPart objects as arguments.

Sample Code: Add an Attachment

```
 // Add a document object attachment.
 msg.attach(doc);
```

Send the Message

To send the message, use the oracle.ifs.adk.mail.IfsMessage.send() method, with no arguments. The only validation the send() method performs is to be sure that one of the recipient fields is set (To, CC, or BCC). All other validation is performed by the outgoing e-mail protocol. The Outbox agent will automatically pick up messages from the outbox. The Outbox agent passes all messages to the SMTP port on the local machine.

Sample Code: Send the Message

```
msg.send();
```

Note that this method will return the following error messages if the message cannot be sent:

- 11031: Either From or Sender header must be present.
- 11032: Either To, Cc or Bcc header must be present.

Sample Code: Sending E-Mail Programmatically

```
package oracle.ifs.adk.mail;
/* --IfsMailTest.java-- */

import oracle.ifs.beans.DirectoryUser;
import oracle.ifs.beans.Document;
import oracle.ifs.beans.FolderPathResolver;
import oracle.ifs.beans.LibrarySession;
import oracle.ifs.common.IfsException;
import oracle.ifs.beans.LibraryService;
import oracle.ifs.protocols.email.beans.Mailbox;
import oracle.ifs.adk.mail.IfsMessage.MimeBodyPart;

public class IfsMailTest
{
  public static void main(String[] args)
  {
    try
    {
      // Setting up the service, session, and user
      LibraryService service = new LibraryService();
      ClearTextCredential me = new ClearTextCredential("tuser1", "tuser1");
      ConnectOptions options = new ConnectOptions();
      options.setLocale(Locale.getDefault());
      LibrarySession session = service.connect(me, options);
```

```
 DirectoryUser user = (DirectoryUser) session.getDirectoryUser();
 if (user.isAdminEnabled())
   session.setAdministrationMode(true);

 IfsMessage msg = new IfsMessage(session);

 // Example 1
 // Creating a simple message being sent to two destinations.
 // The message has a simple string body.
 //
 IfsMessage msg = new IfsMessage(session);
 msg.setToHeader(new String[] {"guest@us.oracle.com",
                               "tuser51@us.oracle.com"});
 msg.setFromHeader("tuser99@us.oracle.com");
 msg.setSenderHeader(user);
 msg.setCcHeader("tuser51@us.oracle.com");
 msg.setBccHeader("tuser51@us.oracle.com");
 msg.setContentType("text/plain");
 msg.setSubject("iFS email API: Mail with body and multiple recipients");
 msg.setBody("Hello there! This is a simple string body");
 msg.send();

 //Example 2
 // Creating a simple message being sent to two destinations.
 // The message has a multipart alternative structure.
 //
 IfsMessage msg = new IfsMessage(session);
 msg.setFromHeader("tuser99@us.oracle.com");
 msg.setSenderHeader(user);
 msg.setToHeader("guest@us.oracle.com");
 msg.setBccHeader("tuser51@us.oracle.com");
 msg.setSubject("iFS email API: Mail with multi-part body");
 MimeBodyPart[] bodies = new MimeBodyPart[2];
 bodies[0] = new MimeBodyPart("hello this is plain text.",
                              "text/plain", null);
 bodies[1] = new MimeBodyPart("<html><body>hello this is a html text " +
                              ".</body></html>", "text/html", null);
 msg.setAlternativeBodies(bodies);
 msg.send();

 //Example 3
 // Creating a simple message being sent to two destinations.
 // The message has a multipart mixed structure.
 //
IfsMessage msg = new IfsMessage(session);
```

```
        msg.setFromHeader("tuser99@us.oracle.com");
        msg.setSenderHeader(user);
        msg.setToHeader("guest@us.oracle.com");
        msg.setCcHeader("tuser51@us.oracle.com");
        bodies = new MimeBodyPart[3];
        bodies[0] = new MimeBodyPart("This is plain text again.",
                                     "text/plain", null);
        bodies[1] = new MimeBodyPart("<html><body>This is html text. " +
                                     "</body></html>", "text/html",
                                     IfsMessage.ASCII_CHARSET);
        bodies[2] = new MimeBodyPart("<html><body>hello this is another html " +
                                     "text.</body></html>", "text/html",
                                     IfsMessage.ISO88591_CHARSET);
        msg.setMixedBodies(bodies);

        //
        // Associate attachments for Example 3.
        //
        FolderPathResolver resolver = new FolderPathResolver(session);
        Document doc = (Document) resolver.findPublicObjectByPath(
                                             "/home/tuser1/test.txt");
        msg.attach(doc);
        doc = (Document) resolver.findPublicObjectByPath("/home/tuser1/test.jpg");
        msg.attach(doc);

        msg.send();
      }
      catch (IfsException e)
      {
        e.printStackTrace();
      }
    }
  }
```

A

Error Messages

This appendix presents typical error messages you may encounter while developing your application. For each message, the cause of the error as well as actions you may take to correct the error condition are provided.

IFS-10170	**Invalid name/credential.**
Cause:	Incorrect Oracle *i*FS login and password were entered.
Possible Actions:	1. Re-enter the correct login and password. 2. Confirm that a user exists in Oracle *i*FS with the specified name.

IFS-10200	**Unable to access object (insufficient privileges)**
Cause:	User tried to access a PublicObject (such as a Document, Folder, etc.) that the user did not have permission to access.
Possible Actions:	1. Have the owner change the permissions (ACL) to allow user to access the object. 2. If the permissions are set correctly so as to prohibit the user from accessing the object, no other action is applicable.

IFS-10406	Invalid AttributeValue conversion ({0} to Java {1})
Cause:	This error generally occurs when writing directly against the Java API. The error is coercing an attribute value to an incorrect datatype. An example of an invalid conversion is coercing a DATE to a BOOLEAN. An example of a valid conversion is coercing an INTEGER to a STRING. One case worthy of a special mention is in converting from PUBLICOBJECT to PUBLICOBJECT. This will fail if the user does not have permission to access the PublicObject referenced in the AttributeValue. The parameters {0} and {1} will have the actual values that caused the error.
Possible Actions:	1. Check the datatypes, and modify them to be compatible datatypes. 2. In the case of coercing to a PublicObject, check whether the user can access the PublicObject referenced in the AttributeValue.

IFS-10600	Unable to construct library connection
Cause:	This error occurs when the Oracle *i*FS repository cannot connect to the database. Generally, this is caused by an invalid database username specified in the service properties file, or an invalid database password. It can also occur if the DatabaseUrl setting in the service properties file is invalid.
Possible Actions:	1. Verify that the database username, password, and TNS names entry are set correctly, by using SQL*Plus or a similar tool to connect to the database. 2. Check to see if this exception encapsulates another exception that describes the cause more clearly.

IFS-10620	Unable to construct connection pool
Cause:	This error generally occurs when database connections cannot be made. Typically, error 10633 causes this error to be thrown.
Possible Actions:	If error 10633 has caused this exception, verify that the database username, password, and TNS names entry are set correctly, by using SQL*Plus or similar tool to connect to the database.

IFS-10633	Unable to create library connection
Cause:	This error generally occurs when database connections cannot be made. Typically, error 10600 causes this error to be thrown.
Possible Actions:	If error 10600 has caused this exception, verify that the database username, password, and TNS names entry are set correctly, by using SQL*Plus or a similar tool to connect to the database.

IFS-12200	Invalid item name specified (<item name>).
Cause:	An attempt was made to look up an object by name in one of the Oracle *i*FS Collections, and no object by that name exists. This can occur when directly invoking the getItems() method on the Collection class, or indirectly by performing an operation that will access one of the Collections. An example of the latter case is when a ClassObject name is specified in an operation such as creating a new Document, and there is no ClassObject with the specified name.
Possible Actions:	Check the name specified, and re-enter a valid name.

IFS-12620	**Parser: syntax error (parameter)**
Cause:	A syntactical error was detected by a parser while parsing a document stream being introduced into the Oracle *i*FS repository. The parameter identifies the token responsible for the syntax error. For example, the SimpleXmlParser will throw this exception when an unknown tag is encountered while parsing an XML file; the parameter is the unknown tag value.
Possible Actions:	Fix the document body, correcting the syntax error, and re-submit to Oracle *i*FS.

IFS-20000	**Unable to get repository parameter (parameter)**
Cause:	This error typically occurs when trying to run Oracle *i*FS against a partially installed Oracle *i*FS instance or a very old Oracle *i*FS instance (e.g., older than 1.0.8.0.0). This is particularly true if the parameter listed is the string "SCHEMAVERSION".
Possible Actions:	Verify that the Oracle *i*FS instance on which this error occurs has been installed properly, and is version 1.0.8.0.0 or higher.

IFS-20001	**Unable to get schema version.**
Cause:	This error is typically caused by error 20000, the inability to get the repository parameter named "SCHEMAVERSION". This generally occurs when trying to run Oracle *i*FS against a partially installed Oracle *i*FS instance or a very old Oracle *i*FS instance (e.g., older than 1.0.8.0.0).
Possible Actions:	Verify that the Oracle *i*FS instance on which this error occurs has been installed properly, and is version 1.0.8.0.0 or higher.

IFS-20010	**Failed to get PropertiesResourceBundler <parameter>**
Cause:	This error occurs when attempting to start an Oracle *i*FS process by specifying a service properties file name that cannot be located by the Oracle *i*FS repository. The parameter specified in the error is the name of the specified service properties file. The specified service properties file must exist in the oracle.ifs.server.properties package descending from one of the directories included in the CLASSPATH setting for the process.
Possible Actions:	Check for the existence of a service properties file with the specified name reachable from the current CLASSPATH.

IFS-21008	**Login failure (2)**
Cause:	An attempt to establish an Oracle *i*FS session has failed, usually because the specified credential (name / password combination) is invalid. In this case, this error encapsulates the error 10170; for all other (rare) authentication failures, error 10150 is encapsulated.
Possible Actions:	If the login failure is caused by invalid credential, re-enter the valid credential to establish an Oracle *i*FS session.

IFS-30002	**Unable to create new LibraryObject**
Cause:	The creation of a new Oracle *i*FS object has failed. The actual cause of the failure is described in an exception encapsulated by this exception. For example, if a uniqueness constraint is violated when attempting to create a new object, the top-most exception will be 30002, and it will encapsulate exception 30010: "Attribute would not be unique (<attribute>)".
Possible Actions:	Investigate the cause of the object creation failure, take corrective action, and retry.

IFS-34611	Error reserving version series
Cause:	An error has occurred in "checking out" a versioned PublicObject, e.g., reserving the VersionSeries object associated with a versioned PublicObject. The failure can be caused by number of conditions, listed below, and described in most cases by inspecting the encapsulated exception. For example, another user already has the VersionSeries reserved, the encapsulated exception will be 34602: "Operation not permitted, version series is reserved."
Possible Actions:	1. Make sure the VersionSeries is not reserved or locked by another user. 2. Make sure that the Family is not locked by another user. 3. Make sure that the last version in the VersionSeries is not locked by another user. 4. Verify the current user has the permission "AddVersion" on the target VersionSeries.

IFS-46113	No such Server (name)
Cause:	An attempt was made to look up an Oracle *i*FS Server by name, using one of the ServerManager interfaces, when no such server exists. This can occur if the name is improperly specified, or when a server that matches this name is no longer running.
Possible Actions:	1. Re-check the active server list, using the ServerManager interfaces; e.g. by using the "list servers" command in the ServerManager commandline interface. 2. Check the ServerManager and/or protocol server logs to see if a server by the specified name has stopped unexpectantly.

IFS-46114	**Server name IfsProtocols is ambiguous; specify the server identifier:(<id>, <id>, ...)**
Cause:	An attempt was made to look up an Oracle *i*FS Server by name in the ServerManager comandline interface, and more than one server exists with this name. The identifiers listed in the exception text are the unique server identifiers of the servers that have the specified name. These identifiers can be used in place of the server name to perform an operation on a server.
Possible Actions:	List the servers using the -i option in the ServerManager commandline interface. Then, re-submit the original server request by using the identifier instead of the server name.

Index