Java Gateway Programmer's Guide

EsiObjects V4.0

ESI Technology Corporation

5 Commonwealth Road

Natick, MA. 01760

www.esitechnology.com

Table of Contents

Introduction

This guide is designed to assist the EsiObjects programmer in using the Java Gateway to build object oriented application systems. It contains the following:

- An overview of the Java Gateway.
- The concepts of the Java Gateway.
- Description of the commands and features added to the EsiObjects Development Enviroment.
- How to perform basic lifespan operations.
- How to use the gateway with proxies.
- Description and how to use Bulk Transfer Objects in conjuction with the gateway.
- How to use event processing through the gateway.
- How to use some advanced features of the gateway.

Document Convention

EsiObjects documentation uses the following typographical conventions:

For more information on this subject please refer to the <u>BREAK Command section of this Guide.</u>	Underlined text is used to highlight a reference to another section of this manual or another guide.
Property	In text, italicized words indicate defined terms that are usually used for the first time. Words are also italicized for emphasis.
CREATE	Words in bold and capitalized are EsiObjects commands or keywords.
`Set T%Test=I%Pat.Name`	This font is used for code examples.

Overview

What is the Java Gateway?

The Java Gateway is a set of Java classes that allows Java applications to access one or more EsiObjects servers. TCP/IP handles communications between a Java client and the EsiObjects server, which is the only method this implementation provides.

The gateway, in conjunction with Java Proxies, allows for efficient and specialized services to the server. These components are all described in this document.

Using the gateway and proxies, a Java client can connect to the EsiObjects server and have access to the basic services in EsiObjects: object creation and destruction, operations on methods, properties, and relationships, events and more.

The client, when connected to the server via the gateway, occupies one M process that handles the communications between Java and EsiObjects. Thus the objects created within this process are valid only in the client that issued them, unless the object is created as a *shared* object (more on this later.)

Scope of this Document

This document describes the basic usage of the Java Gateway and gives examples using JDK 1.3 using Forte. Using this guide you will learn how to:

- Connect and disconnect a Java client to an EsiObjects server.
- Create Java Proxies from EsiObjects that are included in your Java application. These proxies represent the actual objects on the server.
- Lookup, create and destroy objects.
- Use Proxy objects on the client to invoke methods, set and get property values, and set relationships on server objects.
- Use event processing.
- Use bulk data transfer objects that are supplied with the gateway to allow the transfer of bulk data across from the server for manipulation on the client, and then back to the server.

Java documentation is included with the Gateway that describes the *com.esitechnology.eo* package.

The Java Gateway Package

When you install the Java Gateway from the installation kit, several components will be placed on your system.

- EsiObjectsV4.JAR – The JAR file of the Java Gateway classes – required for runtime.
- EsiObjectsV4Doc.Zip – The Java Docs for the Java Gateway.

What Else You Will Need

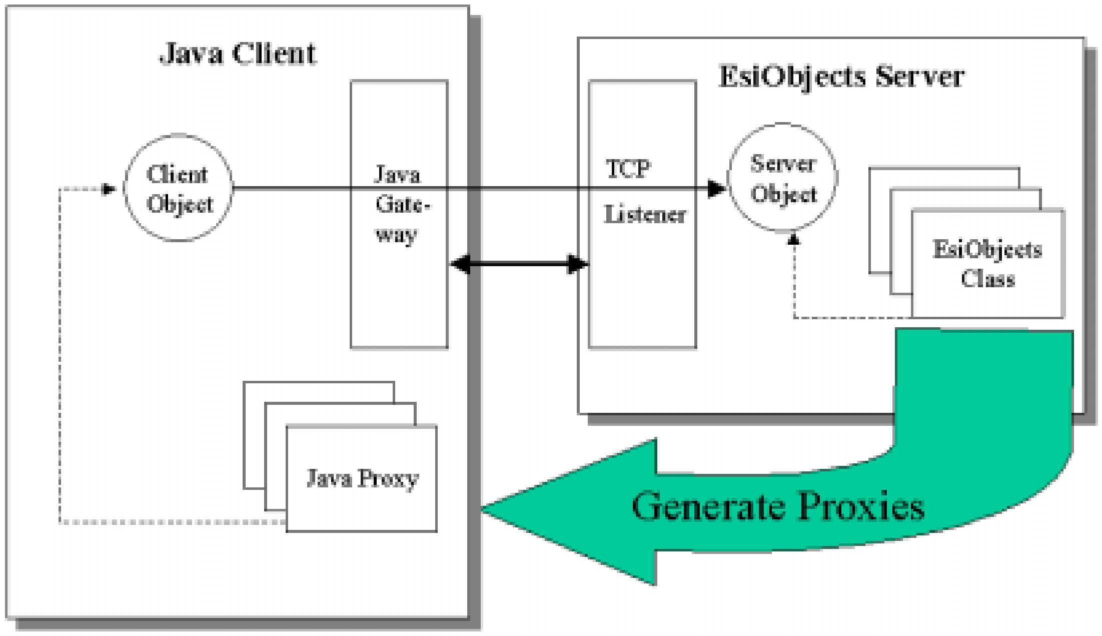
- JRE – Java Runtime environment, version 1.2 or better.
- EsiObjects Server – A running EsiObjectsV4.0 or better.

Concepts

Communications Diagram

The diagram below shows how the Java Gateway relates to the EsiObjects server. EsiObjects classes that define objects to be accessed from the client have Java Proxies generated for them. This process produces a java file for each generated class. The class files are included in the client application. Also included is the java gateway package. At runtime, the gateway is connected to a TCP Listener process running on an EsiObjects server. Once connected, the Java programmer can access the EsiObjects server via the gateway. The gateway is the object that routes requests from the client to the server. A Java Proxy object maps to a server object. Operations invoked on the proxy are sent, via the gateway, to the servant object. The operation is executed on that object any result is returned back to the proxy.

Not shown in the diagram are the Bulk Transfer Objects (BTO) that the gateway provides for retrieving and sending collections of data to and from the server. These objects are described later in this guide.



The gateway and generated java proxies are included in the Java client. The gateway uses TCP/IP to connect to a TCP listener that is running on the EsiObjects server. Proxies are used to invoke operations on objects within the server.

The Java Gateway, and Generated Proxies

The Java Gateway

The Java gateway is a set Java classes that provide a communications link to the EsiObjects server. Once connected to a server, the gateway provides services for creating and looking up objects. These objects are created as proxy objects on the java side and are connected to a corresponding object on the server side. These proxies use the services of the gateway to connect to the server object. Methods, properties, and relationships of these objects are handled via the gateway that provides the connection from client to server.

Besides handling the connection to the server, and routing requests from the proxy object to the server object, the gateway has a set of common support classes such as Variant and bulk transfer objects (BTO) that provide services for moving large amounts of data between client and server.

The classes provided in the gateway are all extensible by the user.

Generated Proxies

From within EsiObjects, a user generates a Java Proxy class from EsiObjects class definitions. These proxies are java class files that contain the methods, properties, and relationships defined in that class. You can generate only the Java interfaces for each of the services defined in the class, or the implementation code as well. The implementation file contains all the necessary functionality to communicate from the client to the server object since the connection to the gateway is automatically handled within the generated proxy code.

Overview of the com.esitechnology.eo Package

The following sections deal with the contents of the gateway package – namely the com.esitechnology.eo package. The information presented here is a basic overview. Detailed reference information can be found in the Java Doc provided with the gateway.

Major Classes Used by the User

TCPGateway

This class is the primary implementation of the Gateway interface. It provides the main communications component for talking to the EsiObjects Server.

Variant

When data is passed to and from EsiObjects, there needs to be a consistent and known representation of the data. For this purpose, the Variant class is used. It acts as a generalized data holder. The Variant data type is the default parameter and return data type.

EoEventListener

The gateway provides for events. This means a Java client can "watch" objects in the server and be notified of events that have occurred. This class is the generalized definition for an Event Listener that is responsible for responding to events.

EOEvent

When an event occurs in EsiObjects, the data corresponding to that event is passed to the client in an EOEvent structure.

EOPropertyChangeEvent

In EsiObjects, properties can be watched for changes. Thus, when a property is modified, EsiObjects throws an event that watchers receive. This event is generated by the server as an EOPropertyChangeEvent.

SimpleClientServices

EsiObjects can provide some client services such as asserting a message box to the client display, etc. These basic services are provided in this class.

Subclasses of Interface Base – Generated by EsiObjects.

All Proxy classes generated are subclasses of InterfaceBase class.

BTO Implementations

Bulk Transfer Objects enable the passing of data from clients to servers and back. Below is a brief description of each of the BTOs available to the programmer.

List

A List is a simple sequential collection of Variants. It provides the implementation of BTOList interface.

NvList

The NvList provides a list of named items. It provides the implementation of BTONvList interface. Each item in the list may be given a name – also called a "key". Names need not be unique. An index based on name is maintained which allows items to be retrieved by name. An NvList object can also be used like a Map that maps names to values. The NvList is implemented as a sequential collection of Variants; each one of them may be named.

Table

A Table is a collection of data organized by rows & columns. It is optimized for row based operations more than column based lookups, etc. Table provides the implementation of BTOTable interface.

In Java, the Table is implemented as a collection of Variants arranged into a table. The Table supports names columns.

To be useful the Table should always be dimensioned especially the number of columns in the table. Although it is legal to specify zero for one of the two dimensions, this is not recommended. Rows and columns are zero-based.

Text

The Text object supports very long block text. Given the restrictions of most M systems, large blocks of text must be store in a collection of blocks. The Text object is the client side representation of the ESI$Text object. It provides the implementation of BTOText interface.

The operations on Text facilitate the breaking of the large text object into smaller chunks that can be stored in M.

NamedCell

This object mimics a cell in an NvList. It has two fields: Name and Value. Methods on the NvList can make use of a NamedCell object to set and retrieve an element from an NvList.

Exceptions

The following are the additional exceptions that are used by the gateway.

NoConversion

When converting data, especially from within a Variant type to a specific type, this error is generated when data cannot be converted to the requested type. This exception extends *Error* and thus this exception need not appear in a catch block.

RequestFailed

This exception indicates that a proxy service request failed.

Interfaces

CoreObject – defines the required services for a proxy object.

Gateway – General definition of the gateway service available in EsiObjects.

BTO – Generic interface implemented by all EsiObjects Bulk Transfer Objects. Provides the core services needed for identification wire transport form.

BTOList – Defines the interface for a simple list.

BTONamedCell – A named data item. NvLists are collections of these.

BTONvList – Interface to a list of items that are named.

BTOTable – Interface to a Table collection containing elements arranged by rows and columns.

BTOText – Interface for a string of text.

ClientServices – Standard Client Side services requested by the Server.

EoEventListener – Interface for an Event Listener.

OutputHandler – Abstraction of an Output Service that the server may call. **Standard Pane Ids**: 0=General Output, 1=Build, 2=Debug.

StatusHandler – Interface for a Status Handler (a Wait Window.)

StatusHandlerFactory – Interface to create Status handlers.

DebugSink – Interface for debugging related service callbacks. This defines the basic interface that the EsiObjects Service will invoke while debugging.

DebuggingServices – Interface to the debugging service provided by a gateway.

Implementation Classes

DataTransferTracker – The DataTransferTracker allocates and track Id strings for objects to give these objects a reversible identifier.

EventManager

InterfaceBase – Base class for all interface objects to EsiObjects. **Note:** All method names start with "*javaProxy_*" to avoid conflicts with the actual method names in EsiObjects.

Parameter – Abstraction of a Parameter to be passed into EsiObjects.

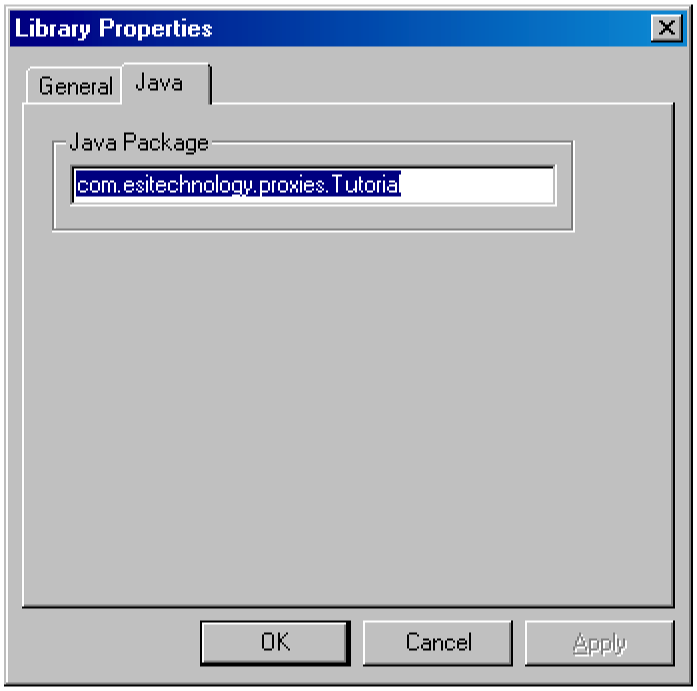
WeakValueMap – Used to track EsiObject Object Id and associate them to the equivalent proxy object. The use of the class will insure that the mapping alone does not keep the proxy alive.

TCPDebuggingServices – Implementation of the TCP Debugging services.

User Interface

Class Library Properties Page

Right clicking on a library name in the EsiObjects Session Browser window will bring up a Property Sheet dialog for the library. This dialog contains two pages. The 2nd tab “Java” contains a single prompt for the name of the Java package in which proxies from this library will be created.



Java Package: com.esitechnology.proxies.Tutorial

The Java page for the Class Library Properties dialog.

When you generate a proxy for any class in this library, the package that the proxy will be created in will be whatever is entered here. For example, when generating a Java Proxy for a class in the Tutorial library, the top of the .java file will contain the following entry:

```
package com.esitechnology.proxies.Tutorial;
```

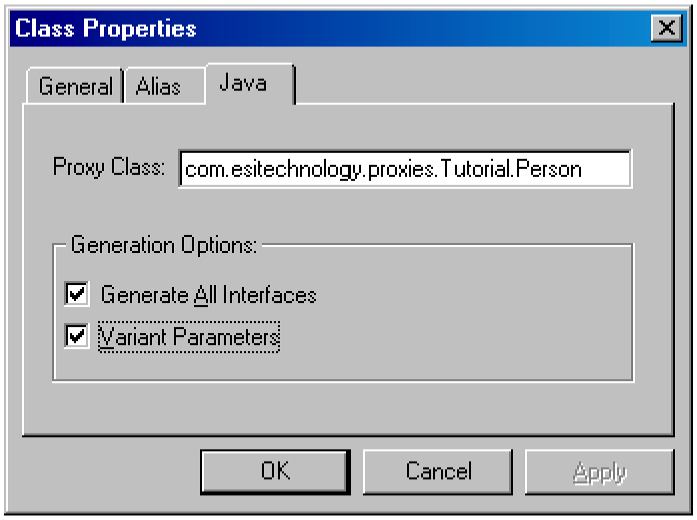
Whatever name was entered in the Java Package field on this dialog will be used as the package name in the generated .java file.

Note: The default package name is displayed in the Java Package field until it is changed. Setting this field to null will restore the default value.

Class Properties Page

Similar to a Class Library, a property dialog can be brought up for a class by right-clicking on the class name in the Session Browser window and selecting *Properties*.

The following dialog is displayed:


Class Properties

Proxy Class: com.esitechnology.proxies.Tutorial.Person

Generation Options:

[✓] Generate All Interfaces

[✓] Variant Parameters

The Class Properties Dialog with the <u>Java</u> page selected.

Selecting the *Java* tab allows you to specify three fields:

Proxy Class

This is the location of the proxy that will be generated. The default value for the Proxy Class field is the value of the package name entered for the class library (see above) with the name of the class appended. This default value remains here until changed. Setting this field to null will reinstate the default value.

Generate All Interfaces

Selecting this option (the default) will generate code for all services in all interfaces. Un-selecting this field will cause the java proxy to be generated with Primary interface services only.

Variant Parameters

By default parameters on the Java generated code will be of type Variant. Un-selecting this field will cause all parameters to be of type String.

Example:

Here is an example of a generated code for the Person class in Tutorial Library

Variant Parameters:

```
public abstract Variant getAddress() throws RequestFailed;
public abstract void setAddress(Variant Param1) throws
RequestFailed;
```

Without Variant Parameters:

```
public abstract Variant getAddress() throws RequestFailed;
public abstract void setAddress(String Param1) throws
                     RequestFailed;
```

Note the return types are always *Variant*. But the parameter type changes from Variant to String depending on the setting in the Class Properties dialog.

Generate Java Proxies Interface

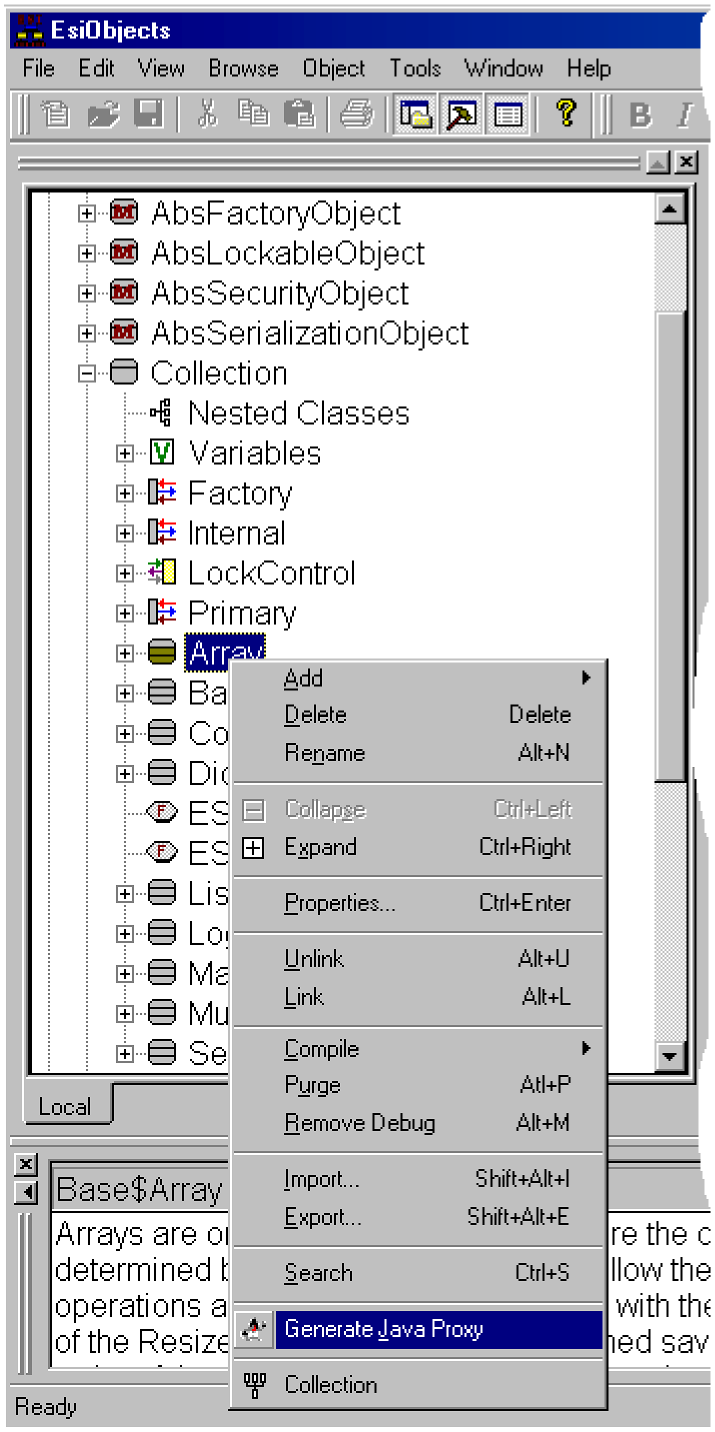
You can use the default values in the Class Library and Class property pages or modify them for your needs. The next step is to generate the java proxy itself. The *Generate Java Proxy* option is available from several areas:

Popup menu on a class – this will allow you to generate a proxy just for the selected class.

Popup menu on a class library – this will generate proxies for all classes in the library.

From the Tools | Java menu. **–** generating proxies from this option will generate a proxy for whatever is currently selected in the Session Browser panel. If a class is selected, then a proxy for that class is made. If a library is selected, then all classes in that library will have proxies generated from them.

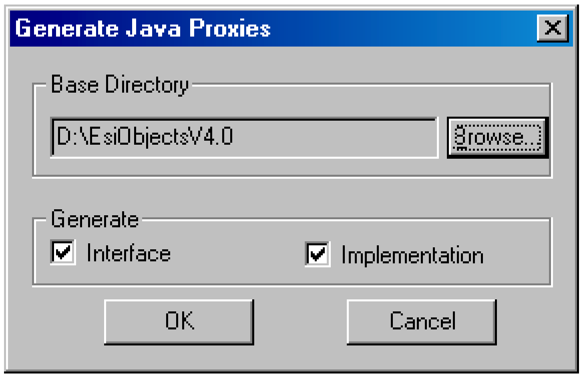
Below is illustrated the Generate Java Proxy option from the Class popup menu. This menu is displayed when you right-click on a class name.



Generate Java Proxy option from the Class popup menu

Similarly, right clicking on a Class Library name in the Session Browser will also bring up a menu that includes this option.

In each case, when you select the Generate Java Proxy option, the following dialog is displayed.



D:\EsiObjectsV4.0

[✓ Interface] [✓ Implementation]

The Java Proxy Dialog

Base Directory – specify the base location under which the java package/proxy code will be placed.

Generate Interfaces – selecting this will generate interface signatures for services implemented in the class.

Generate Implementations – selecting this will generate implementation code for the proxies.

The file(s) generated are:

- *Classname*Impl.java – for implementation code
- *Classname*.java – for the interface

For example, if we generate both interface and implementation code for the class Tutorial$Person, using a base directory of D:\EsiObjectsV4.0 and default package names, two files would be created:

```
D:\EsiObjectsV4.0\com\esitechnology\proxies\Tutorial\P
erson.java
D:\EsiObjectsV4.0\com\esitechnology\proxies\Tutorial\P
ersonImpl.java
```

The first file is the interface file and the second is the proxy implementation code.

Basic Operations

Once you have generated the proxies for classes you wish to make use of in your Java application, you need to incorporate these proxies and the gateway into your client. The following sections describe how to include the gateway and proxies in your client and the types of operations available to you via the gateway. This includes connecting to an EsiObjects server, creating objects within the server from your client and invoking services on them.

Getting Started

The following steps will get you started in hooking up your Java client to EsiObjects using the Gateway.

Installing and locating the EsiObjectsV4.JAR

In order to use the Java Gateway the EsiObjectV4.JAR must appear in the users Class path. Depending on how you are developing the mechanism for doing this may vary.

- In JDK 1.2 the environment variable CLASSPATH could be used.
- In all JDK versions there is a command line qualifier on the Java Compiler (JAVAC) to specify the Class path.

Most development environments have a mechanism for specifying the CLASSPATH.

For example, in Forte you can mount the JAR as a File System

For Deployment:

When the user deploys an application written using the Java Gateway they will need to package the EsiObjectV4.JAR file with their product, and they will need to load this file with their application. The end users will need to make sure the JAR file is in their class path for the installed application.

When running an application interactive the JAVA.EXE program is used. It accepts a command line qualifier to specify the class path.

If the application is deployed over the web, the EsiObjectV4.JAR will need to be included.

Import com.esitechnology.eo.*

It is recommend that your Java code should include an import statement for the “com.esitechnology.eo” package. This will simplify the development process. This step is optional, but it omitted full class names number be used. It is also possible to import only selected classes from the package.

Example:

```
//Import the entire package
import com.esitechnology.eo.*;
//Imports the selected items from the package
import com.esitechnology.eo.TCPGateway;
import com.esitechnology.eo.Variant;
```

Declare the TCPGateway Object

In order to use the TCPGateway an instance must first be declared with in your application. Depending on wheather, you have imported the com.esitechnology.eo package or not the form of the declariation will vary. In either case the class of gateway is com.esitechnology.eo.TCPGateway.

Example:

```
/* Declaration of TCPGateway when importing the
   com.esitechnology.eo package */
import com.esitechnology.eo.*;
private TCPGateway theGateway;

//Declaration of the TCPGateway when no impoer is done
private com.esitechnololgy.eo.TCPGateway theGateway;
```

Create an instance of a TCPGateway Object

In Java declarion of an object does not create the object, thus it must be created before it can be used. This is done using the new operator.

Example:

```
import com.esitechnology.eo.*;
.
.
.
private TCPGateway theGateway;   // Declare the Gateway
theGateway = new TCPGateway();   // Create the Gateway
//Alternative method Declare & Create the Gateway
private TCPGateway anotherGateway = new TCPGateway();
```

Connection

Users of the gateway connect to the server explicitly. Often, an application looks up local information about what server and port to connect to from its configuration information and then explicitly connect to that server. Some applications will query the user for this information. In any case, the **openConnection** method is invoked to initiate the connection.

Using openConnection()

OpenConnection() will establish a TCP/IP connection to an EsiObjects TCP listener. This method requires a valid a Host Id and Socket Number. Upon conclusion of the call, the Gateway will be associated with a session and can make requests. Note that the connection may well be redirected to another host and port.

Example:

```
try{
  theGateway.openConnection(“appsrv4.esitechnology.com”,9000,””)
}
catch (UnknownHostException exp){
//Handle the exception
}
catch(RequestFailed exp{
//Handle the Exception
}
catch(IOException exp){
//Handle the Exception
}
```

Common Exceptions and Their Causes

The following lists the common exceptions you may get when attempting to connect to the server.

UnknownHostException

- Bad Host Name or Address specified
- There could be a DNS problem
- Requested Address is not reachable from your location

RequestFailed

- Bad Port specified in the parameters
- EsiObjects server listener is not running at requested port
- Server overloaded

IOException

- This indicates a communications error.

Disconnection

To disconnect the gateway from the sever, the **closeConnection()** method is invoked on the gateway.

Example

```
theGateway.closeConnection();
```

Effects of Disconnection

Once the gateway disconnects from the EsiObjects server, any requests made on the gateway will result in exceptions. Similarly, any invocation of services on a Java proxy will also generate an exception.

Bulk Transfer Objects (BTO) will still work since these objects exist completely on the client.

Using lookupObject()

Once a connection is made to the server one of the first tasks is locating objects on the server.

The **lookupObject()** is one mechanism for locating persistent and system objects on the Server.

Locating System Variables

The table below lists the system objects that the **lookupObject()** service may find. The names in the Table are not case sensitive.

System Object	Abbreviation	Description
$ENVIRONMENT	$ENV	The Environment object associated with this connections
$LIBRARY	$LIB	The default Library
$LIBRAYRLIST		The List of all Class Libraries
$SYSPOOL		The System Name Pool

Locating Class Objects

The **lookupObject()** service may be used to find the Class Object associated with a class name. When looking up the class, the name used should be the full class name prefixed with an underscore. For example to find the class object for the class TimeStamp in the Base Class Library the name would be:

```
_Base$TimeStamp
```

The standard format for nested class names is:

```
Lib$Class>Nest1>Nest2
```

Locating an O% Name

LookupObject() may also be used to find named objects in the current default domain. When coding in EsiObjects such names are prefixed with an “O%”. When using the **lookupObject()** service, the O% should not be used. Instead just the name is specified.

If a name is not found then an empty string will be returned. It is thus possible to check to see if name is defined by checking against the empty string.

Examples:

```
//Define the Variants to get the result
Variant environment;
Variant setClass;
Variant database;
Database DB;
//Find $ENV
environment=theGateway.lookupObject(“$Env”);
//Find the Class Base$Set
setClass=theGateway.lookupObject(“_Base$Set”);
//Check if an O%Database is defined
database=theGateway.lookupObject(“Database”);
If  (data .getString().length()==0){
      //   Code to deal with the undefined Database
}
else{
    //The Database is defined lets get it
    DB = (Database)database.getObject();   // extracts the Proxy
object
 // from the variant
}
```

Working with Variants

The last line in the example above shows how to get a handle to a Proxy object from the Variant. The following sections deal with Variants.

What is a Variant

A Variant is a data type that can be thought of as a generic container for primitive data. A variant is a structure that contains 2 fields: one that contains an indicator of what actual type of data is contained in the variant (a Boolean, integer, string, etc) and the second field contains the data.

The types of data that can be placed in a variant are listed below.

- Boolean
- Double
- Float
- Integer
- Long
- Object
- Short
- String

Variant are used extensively by the Gateway. Parameters used on calls into the gateway, by default, are variants. Return data from these calls are also variants.

Creating a Variant

Variant can be created with data seeded into them, or without data. When using the *new* command, a parameter is passed that contains the data to place into the Variant.

Create without Data

```
Variant x = new Variant()
```

Create with Data

```
Variant x = new Variant(dob)
```

Declaring a Variant

Variants can also be simply declared and data set into it.

Example:

```
Variant ret;       // declare a variant
try {
// delete
ret = db.Delete(curPer.getValue());        // fill in with result
 .
 .
 .
Variant v = o[0];        // declared and seeded with data
```

Used in a Argument

When passing a variant as an argument the *new* command is useful. For example:

```
if (rdbMale.isSelected()) nvl.setAt("Sex", new Variant("Male"));
```

Get Data from a Variant

Getting data from a variant is accomplished by requesting the specific datatype. For example, if the variant contains an object, the programmer uses the **getObject()** call. If the variant contains text, **getString()** is invoked.

Examples:

The following example shows a variant being returned from a service call. The method invocation returns a variant data type. The Object is obtained from the variant via the **getObject()** call.

```
Variant v = db.GetOrgNameList(new
Variant(txtSelOrg.getText()+""));
NvList nvl = (NvList)v.getObject();
/* The next example shows how a text string is extracted from a
variant. */
protected String oidEnv = "";
Variant vaRet = null;
vaRet = gw.lookupObject("$ENV");
oidEnv = vaRet.getString();
```

Set Data into a Variant

Setting a value into a variant requires invoking the proper method corresponding to the type of data being inserted. If you are inserting a string:

```
va.setString("Hello");
```

inserting a Boolean:

```
va.setBoolean(0);
```

Refer to the Java Docs for information on what data can be set into a variant.

Clearing & Deleting Variants

To clear a variant for use, the **clear()** method is to be used.

```
va.clear();
```

Deleting a variant is accomplished by setting the variable to null.

```
va=null;
```

Objects in Variants

Variants can also contain objects. The **setObject()** method is used to insert an object. The following objects can be inserted into a variant.

- Objects that implement CoreObject interface.
- Objects that implement the BTO interface.
- Objects that implements Stream interface.
- Other objects such as java objects.

Creating and Destroying Objects (Lifespan services)

Lifespan services are used to create and destroy objects on the server. They allow the client to directly create and destroy objects on the EsiObjects server.

Lifespan services are those services used to create and destroy objects on the server. They allow the client to directly create and destroy objects. This section of the guide deals with creation and destruction of objects using the gateway. There are two methods supplied with the gateway for creating objects: s**impleCreateObject()** and c**reateObject()**. The method d**estroyObject()** is used for object destruction.

In EsiObjects, the format of the Create command is as follows:

```
Create Var=ClassName(param1,…):(keyword=value,…):(property=value,…)
```

Recall that the *params* are those parameters that are passed into the CREATE method in the Factory interface.

SimpleCreateObject() allows for specifying positional parameters, only the SHARE keyword, and no property assignments. C**reateObject()** allows you to make full use of the EsiObjects construction syntax – both positional and named parameters, most creation keywords, and property assignments.

simpleCreateObject()

Syntax: `simpleCreateObject(`*`classname, flags, params`*`)`

This method handles simple cases of object construction. The method returns an Object within a Variant. Only *classname* and *flags* are required. Use this method if you wish to create objects without using named parameters. It does allow for creating shared objects. In summary, simpleCreateObject():

- Is used to create objects with out special construction requirements.
- Supports creation of shared objects.
- Does not support specifying FIXED or BASE locations for the object.
- Supports positional parameter passing only by using an array of Variants.
- Does not support specifying properties for the construction of the object.

Use createObject() if other creation features are needed, such as named parameters or property assignments.

The method accepts three parameters:

- The class name (string) for the object you want to create. You should use the fully qualified class name, which consists of the class library name along with the class name, e.g. “Base$Array”.
- A flag (long) indicating whether the object is to be a child (0) or a shared object (1).
- An array of Variants (may be empty) for positional parameters for the Factory::CREATE method of the class.

Examples:

```
#import com.esitechnology.proxies.SuperBase.Query;
#import com.esitechnology.proxies.Corp.Person;
Query query;
Variant [] params;
/* Create a temporary object of the hypothetical class
SuperBase$Query. Creating an object of this type requires three
creation parameters
*/
params = new Variant[3];
//  The database to use for the Query
params[0] = new Variant(”MainDatabase”);
//  The UserName
params[1] = new Variant(”Jones, Fred C”);
//  The Encrypted Password
params[2] = new Variant(”LKE028833HGX2”);
```

```
query = (Query )theGateway.simpleCreateObject( Superbase$Query”, 0,
params);
/* When we are done with the object we should destroy it.
Create a Shared object and tell a security tracker object about
it.
*/
Person person;
//We only have a single parameter on the call
params = new Variant[1];
params[0] = new Variant(”Jones, Fred C”);
person=(Person) theGateway.simpleCreateObject(“Corp$Person”, 1,
params)
securityTracker.RegisterPerson(Person);
```

createObject()

Syntax: `createObject(`*`classname, flags, params, namedParams, options, properties`*`)`

This method allows objects to be created using a broad range of functionality. It supports almost every aspect of the EsiObjects Create command. The method returns an Object. In addition to the functionality provided by s**impleCreateObject()** this method supports:

- Supports most creation keywords.
- Supports Named Parameter passing.
- Supports creation-time property assignment.
- Many of the parameters are optional, they be omitted or an empty collection passed in their place.
- Positional parameters are passed as a BTOList.
- Named Parameters are passed as a BTONvList. The Value of each element in the list is the value of the parameter, while the Name of the element is used as the parameter name.
- Creation Options are passed as a BTONvList. The Name of each element is the creation keyword (case sensitive), while the value of the Element is the value of the creation keyword.
- Creation Properties are passed as a BTONvList. The Name of each element is the name of the property, while the Value of the element is the value to be assigned to the property.

Examples:

```
#import com.esitechnology.proxies.Local.OrderServer;
OrderServer  orderServer;
List parameters = new List();
NvList options = new NvList();
NvList namedParams  = new NvList();
NvList properties  = new NvList();
//We are going to create an object of a class “Local$OrderServer”
```

```
/* We create this class in the default domain, with a name of
OrderServer. To do this will need to define two Create options:
*/
//Using the default domain
options.setAt("Domain",new Variant("")) ;
//The Name "OrderServer"
options.setAt("Name","new Variant("OrderServer"));
/* In order to Create an Object in the server we specific an few
parameters. The first two parameters may be passed positionally,
so we set them in the Parameters Object */
//Set the size of the parameter object
parameters.setDimension(2);
//The User Name
parameters.setElement(0,new Variant("Jones, Mark"));
/* The creation also requires an authentication token, which will
be passed as a NamedParameter */
//Pass the Token we have already defined. (In a variant)
namedParams.setPair(-1,"AToken",Token);
parameters.setElement(1,new Variant("All"));
/* We also want to define a few properties of the object when it
is created */
properties.setAt("MaxClients",new Variant(10));
properties.setAt("EnableEvents",new Variant(1));
//Ok we are now ready to Create the Object
orderServer = (OrderServer) theGateway.createObject(
"Local$OrderServer", 0, parameters, namedParams, options,
properties);
//Example of the Same creation without Properties
orderServer = (OrderServer) theGateway.createObject(
"Local$OrderServer", 0,
parameters, namedParams, options, null).
//Example without named parameters or properties
orderServer = (OrderServer) theGateway.createObject(
"Local$OrderServer", 0,
parameters, null, options, null).
```

destroyObject()

Syntax: `destroyObject(`*`objectId`*`)`

This method destroys the server object corresponding to the *objectId*. It is equivalent to the Destroy command in EsiObjects. Keep in mind that in EsiObjects, when using Destroy the object may not always die. The object may have their reference count decremented instead, or they may reject the attempt altogether.

When destroying the object, you can use the following forms to specify the object id:

- Proxy

- String
- Variant

Examples:

```
//Define three different object forms
InterfaceBase obj;      //Proxy
String objectOID;       //String
Variant temp            //Variant
//Get the object in various ways
obj = theGateway.Lookup(“ObjectToKill1”).getObject();
objectOID = theGateway.Lookup(“ObjectToKill2”).getString();
temp = theGateway.Lookup(“ObjectToKill3”);
//Ok now delete it
theGatway.destroyObject(obj);
theGatway.destroyObject(objectOID);
theGatway.destroyObject(temp);
```

Proxy Usage

The main function of the Gateway is to allow a Java client to invoke methods, properties, relationships, etc. on objects within the server. Proxy objects on the client allow this to happen.

A Class in EsiObjects is a blueprint for the state and behavior an object will have. A Class defines the interface to the object – namely the method, properties, relationships and events that an object of that class will provide. EsiObjects supports the ability to generate Java code for any Class defined in EsiObjects. Generating the Java code places the interface of the object in Java code that can then be included in your Java application. This generated code becomes a Java Class definition for an object called a *Proxy*. All the necessary supporting code to enable the proxy to communicate to the server via the Gateway is also embedded into the generated code.

On the client side, Java can create instances of these Proxies like any normal Java object. Behind the scenes, the Java uses the gateway to invoke the creation of the server object that becomes attached to the client side proxy.

Since the generated code includes all the services provided by the server object, the client can invoke these services in a transparent fashion. The proxy object uses the gateway to invoke the service on the server object to which it is attached.

Concepts

Generation (Java)

As detailed above, the Java code is generated from within EsiObjects. When selecting this option you may generate only the interface file or you may also generate the implementation file. The implementation file includes all the necessary supporting code to attach the client side Proxy object with a server side object.

Parentage & Inheritance

Proxies are really two parts, the interface that extends the same hierarchy as found in EsiObjects, and the actual proxy implementation that generally extends InterfaceBase and implements the proxy interface.

Because the generated proxy code includes the services in the EsiObjects class and also those inherited by InterfaceBase, all methods from InterfaceBase begin with *javaProxy_* to avoid possible name conflicts with the names from EsiObjects.

Structure

When invoking operations on a proxy, you have available to you all the methods, properties, and relationships on the objects you generated proxies for. Invoking these services is similar to invoking any service on a Java object: O*bject_id.Servicename* for invoking services in the Primary interface. If you wish to invoke services in other interfaces, prefix the service name with the interface name and underscore (_).

The format of the Servicename is explained below.

Methods

If the service is a method the name of the method is used.

Properties & Relationships

For property and relationship invocations, the format of the service name is get*Servicename* if you want to get the current value of the property or relationship. If you want to set the value, then the format is set*Servicename*.

Examples

Assuming that PerProxy is a proxy object for a server object of type Tutorial$Person, and a Person object has properties for Sex, and Address (among others), we can invoke the value accessor for the property in the following way:

```
String s = PerProxy.getSex().getString();
```

Note that the **getSex()** call returns a Variant and thus the **getString()** call extracts the data in the variant as a string value.

The parameters for setting the value depend on how you generated the proxy. If you generated the proxy with the Variant Parameters setting to TRUE, then you must use variants for the parameter. Otherwise a string value may be used.

```
// Variant parameter
PerProxy.setSex(new Variant(“Male”));

// string parameter
PerProxy.setSex(“M”);
```

The following example shows an invocation of a method on a proxy for an object of type Tutorial$Organization.

```
NvList pers = (NvList)OrgProxy.EmployeeList().getObject();
```

OrgProxy.EmployeeList invokes the method “EmployeeList” in the Primary interface of the object. Since a variant is returned, the **getObject()** method is called to extract the object handle from the variant. This handle is then cast to an NvList datatype.

The following example shows that a method ResetTestDB is being called in the Initialize interface for this proxy object. The proxy "db" is of type Tutorial$Database.

```
db.Initialize_ResetTestDB();
```

Future plans

Currently property Value and Assign accessors are the only ones that can be invoked via the proxy. In the future, other accessor methods may become available to proxy invocation.

Events

Proxy objects can also be used for event processing. A Java client can watch an object in EsiObjects by making use of the event related services.

When generating a Java Proxy for a class, all events, relationships and properties have corresponding event services created for them. For example, in the class Tutorial$Database, there are many events such as "Deleted", "StoreRecord", etc. Also any property or relationship can also be watched such as OrganizationDB.

Each of these has two event calls generated for each. **add*ServiceName*Listener()** and **remove*ServiceName*Listner**. Where ***ServiceName*** is the name of an event, property or relationship. Both opertions take a single argument: an EOEventListener. The EOEventListener provides the "callback" method to be invoked on the client whenever the specified event occurs in the server.

add*ServiceName*Listener()

Invoking this method on a Proxy places a "watch" on the server object for the event specified by the ***ServiceName***.

remove*ServiceName*Listener()

To remove a watch on an object this method is called on the proxy.

Example

The following code places a watch on an object for the *StoreRecord* event.

```
db.addStoreRecordListener(new EoEventListener() {
public void eventOccurred(EOEvent evt) {
dbStoreRecordEventOccurred(evt);
}
});
```

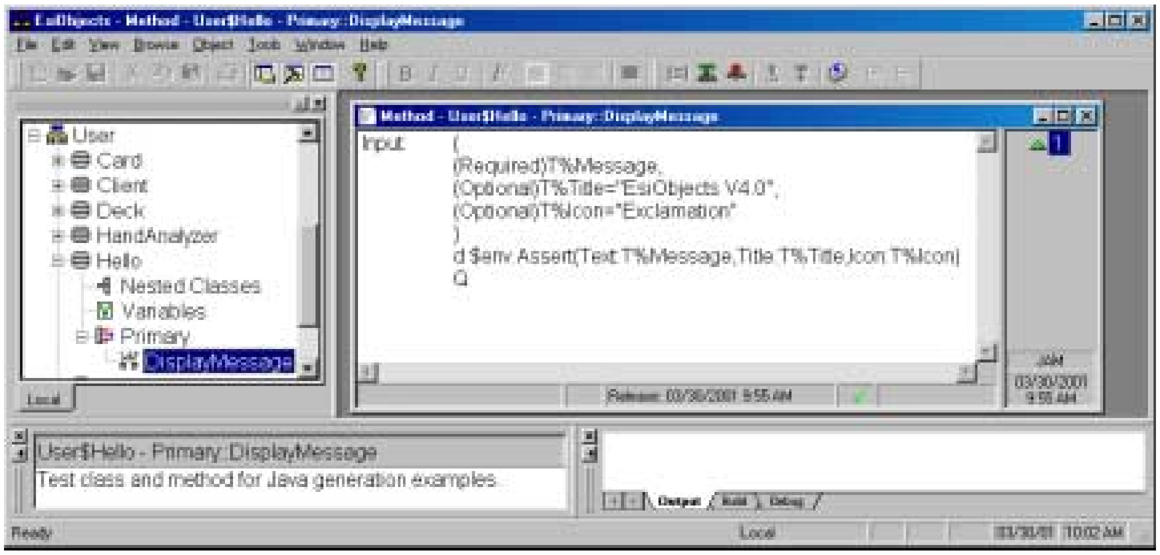
The section *Event Processing* below describes in more detail about how to use events in the gateway.

Comparing

To determine whether two proxy objects point to the same object, the "==" test should be used for comparison.

Example Class

For this example, we created a class "Hello" in the User class library. This class contained one method in the Primary interface called "DisplayMessage".


EsiObjects - Method - User$Hello - Primary::DisplayMessage
File Edit View Browse Object Tools Window Help
User
Card
Client
Deck
HandAnalyzer
Hello
Nested Classes
Variables
Primary
DisplayMessage
Local
Method - User$Hello - Primary::DisplayMessage
Release: 03/30/2001 9:55 AM
03/30/2001 9:55 AM
Output
Build
Debug
Ready
Local
03/30/01 10:02 AM

```
Input   (
        (Required)T%Message,
        (Optional)T%Title="EsiObjects V4.0",
        (Optional)T%Icon="Exclamation"
        )
        d $env.Assert(Text:T%Message,Title:T%Title,Icon:T%Icon)
        Q
```

User$Hello - Primary::DisplayMessage
Test class and method for Java generation examples.

The method DisplayMessage in class User$Hello

This method will display a message box on the screen with a specified message. The method accepts three parameters: the text of the message, the title for the message box, and the type of icon to display in the message box. Only the first parameter is required. The other two have default values if not specified.

Next we generate a Java Proxy for this class, generating both the implementation and interface files.

EsiObjects generated two files in the directory specified in the package path (set in the Properties page of the class.) **Helloimpl.java** and **Hello.java** are the files containing the implementation code and the interface respectively.

Below is the code that was generated for the interface file.

```
package com.esitechnology.proxies.User;
import com.esitechnology.eo.*;
/** Interface to the EsiObjects Class Hello */
public interface Hello  extends CoreObject
{
public abstract Variant DisplayMessage(Variant Param1) throws
RequestFailed;
public abstract Variant DisplayMessage(Variant Param1, Variant
Param2) throws RequestFailed;

public abstract Variant DisplayMessage(Variant Param1, Variant
Param2, Variant Param3) throws RequestFailed;

}
```

Note that even though the EsiObjects class contained only 1 method, DisplayMessage, the generated proxy offers three of these methods.

This is because of the optional parameters that can be passed in this method. If an EsiObjects method has input parameters, the Java proxy will contain a corresponding method for any required parameters. Any optional parameters will have a separate method generated for each permutation of those parameters.

Thus in the example above, the first DisplayMessage was generated with the required parameters – in this case just the one. The second and third generated methods are for the two optional parameters.

Only positional parameter passing is allowed when invoking services on Proxies. No keyword parameter passing is supported.

Limitations

Named Parameters on method invocations are not supported in the Gateway.

Strong data typing is not implemented in this version of the gateway. It may be implemented in future versions.

Future

In the future it may be possible to generate Java proxies that have stronger typing. These proxies will have specified returns types and parameter types based on what is on the EsiObjects server. To some degree this will work hand in hand with the addition of stronger typing in EsiObjects which is also planned for future development. For example one near term change will be to have the relationships generate (and use) their known type information.

For example, given a *Class1* with a relationship *Rel1*. This is a unary relationship to an instance of *Class2*:

The proxies now generate:

```
  public Variant getRel1();
  public void setRel1(Variant value);
```

With strong typing the following will be generated:

```
  public Class2 getRel1();
  public void setRet1(Class2 value);
```

Programming with Generic Proxies

Background

Generic proxies provide the ability to access objects that do not have a proxy available on the client. Depending on how the Gateway is configured, Generic Proxies can be used to supplement a client which only has a partial set of proxies generated.

Platform Issues

Proxy Modes

TCPGateway Constant	Description	Value
PROXYMODE_NO_GENERIC	Generic proxies are never used.	0
PROXYMODE_ALLOW_GENERIC	Generic proxies are loaded if the normal class proxy is not found.	1
PROXYMODE_ONLY_GENERIC	Always use a Generic Proxies.	2

Java Gateway

The Java Gateway supports all three proxies modes. By default the Java Gateway will startup in PROXYMODE_ALLOW_GENERIC. The Proxy Mode may be controlled using the getProxyMode and setProxyMode methods of the Gateway.

PersonalJava Gateway

The Persona Java Gateway differs from the Full Java Gateway in that it only supports Generic Proxies. The getProxyMode method of the PersonalJava Gateway will always return a value of PROXYMODE_ONLY_GENERIC. Since the Personal Java Gateway is designed to work in low memory environments it was engineered to use only Generic Proxies, thus saving the local memory that would otherwise be required for the generate proxies.

Classes

GenericProxy

The GenericProxy provides a generic interface for accessing objects. In order to call a service the user must provide the Service Name and a Parameter Array of the Argumemts.

Methods

Method	**Description**
addEventListener	Add an Event Listener.

addPropertyListener	Add a Property Listener.
Invoke	Invoke a Method.
propetyDataFn	Invoke $Data operation on a Property.
propetryGet	Get a Properties Value.
propetryGetFn	Invoke $Get operation on a Property.
propetryKill	Invoke Kill operation on a Property.
propertyNormalizeFn	Invoke $Normalize operation on a Property.
propertyOrderFn	Invoke $Order operation on a Property.
propertyQueryFn	Invoke $Query operation on a Property.
propertySet	Set a Property.
propetyValidateFn	Invoke $Validate operation on a Property.
removeEventListener	Remove an Event Listener.
removePropertyListener	Remove a Property Listener.

ProxyHelper

Working with the GenericProxy can be cumbersome; the ProxyHelper class provides a simplified façade for accessing the proxy. In order to invoke a service the user must provide a Service Make and the Arguments as Strings or Variants.

Methods:

Method	**Description**
Invoke	Invoke a Method
propertyGet	Get a Property
propertySet	Set a Property
setProxy	Associate the ProxyHelper to a GenericProxy.

Examples:

The following is a set of examples that show how Generic Proxies differ from "Normal" proxy calls. The examples as simplified, and do not include any exception handling. The examples assume a the following interface for a generated proxy:

```
package com.esitechnology.proxies.RGTest;

import com.esitechnology.eo.*;
import java.beans.PropertyChangeListener;

/** Interface to the EsiObjects Class ProxyExample */
public interface ProxyExample  extends CoreObject
{
    public abstract Variant Method1() throws RequestFailed;

    public abstract Variant Method1(Variant Arg1) throws RequestFailed;

    public abstract Variant Method1(Variant Arg1, Variant Arg2) throws RequestFailed;

    public abstract Variant Method2(Variant Arg1) throws RequestFailed;

    public abstract Variant Method2(Variant Arg1, Variant Arg2) throws RequestFailed;

    public abstract Variant getProperty1() throws RequestFailed;

    public abstract void setProperty1(Variant Value) throws RequestFailed;
    public abstract void addProperty1Listener(java.beans.PropertyChangeListener listener);

    public abstract void removeProperty1Listener(java.beans.PropertyChangeListener listener);

    public abstract Variant getProperty2(Variant Arg) throws RequestFailed;

    public abstract void setProperty2(Variant Value, Variant Arg) throws RequestFailed;
    public abstract void addProperty2Listener(java.beans.PropertyChangeListener listener);

    public abstract void removeProperty2Listener(java.beans.PropertyChangeListener listener);

    public abstract Variant Secondary_Method3(Variant Arg1) throws RequestFailed;

}
```

Examples Using Proxy

```
//Preexisting variables
//   result – a Variant with a proxy object in it
//   listener – a java.beans.PropertyChangeListener.

//Declare the Proxy
ProxyExample obj;
//Get the proxies from prior call that stored the result in a variant (result)
obj=(ProxyExample) result.getObject()

//Declare a few variants for parameters
Variant va1=new Variant();
Variant va2=new Variant();

//Declare a Variant for the return value
Variant callResult;

//Call Method1
va1.setString(“Test Value”);
va2.setString(“Another String”);
callResult=obj.Method1(va1,va2);

//Call Method2
va1.setInt(10);
```

```
va2.setInt(300);
callResult=obj.Method2(va1,va2);

//Get Property1
callResult=obj.getProperty1();

//Set Property2

Variant value=new Variant();
value.setString(“Test Value:);
va1.setString(:”Key”);

obj.setProperty2(value, va1);

//Call Method 3 in the Secondary Interface
va1.setString(“A String Value);
callresult=obj.Secondary_Method3(va1);

//Add A Property Listener for Property2
obj.addProperty2Listener(listener);

//Remove a Property Listener
obj.removeProperty2Listener(listener);
```

Example Using a Generic Proxy

```
//Preexisting variables
//   result – a Variant with a proxy object in it
//   listener – a java.beans.PropertyChangeListener.

//Declare the Proxy
GenericProxy obj;
//Get the proxies from prior call that stored the result in a variant (result)
obj=(GenericProxy) result.getObject()

//Declare a few variants for parameters
Variant va1=new Variant();
Variant va2=new Variant();

//Declare a Variant for the return value
Variant callResult;

//Declare a Parameter array
Parameter[] params;


//Call Method1
params=new Parameter[2];
params[0]=new Parameter(“Test Value”);
params[1]=new Parameter(“Another String”);
callResult=obj.invoke(“Primary::Method1”,0,params);

//Call Method2
va1.setInt(10);
va2.setInt(300);
params=new Parameter[2];
```

```
params[0]=new Parameter(va1);
params[1]=new Parameter();
params[1].data=va2;
callResult=obj.invoke("Primary::Method2",0,params);

//Get Property1
callResult=obj.getProperty("Primary::Property1",0,null);

//Set Property2

Variant value=new Variant();
value.setString("Test Value");
params=new Parameter[1];
params[0]=new Parameter("Key");
obj.setProperty(value,"Primary::Property2",0,params);

//Call Method 3 in the Secondary Interface
params=new Parameter[1];
params[0]=new Parameter("A String Value");
callresult=obj.invoke("Secondary::Method3",0,params);

//Add A Property Listener for Property2
obj.addPropertyListener("Primary::Property2",listener);

//Remove a Property Listener
obj.removePropertyListener("Primary::Property2",listener);
```

Example Using a ProxyHelper

```
//Pre- exiting variables
//   result – a Variant with a proxy object in it
//   listener – a java.beans.PropertyChangeListener.

//Declare the Proxy
GenericProxy obj;
ProxyHelper helper;

//Get the proxies from prior call that stored the result in a variant (result)
obj=(GenericProxy) result.getObject()
helper = new ProxyHelper(obj);

//Declare a few variants for parameters
Variant va1=new Variant();
Variant va2=new Variant();

//Declare a Variant for the return value
Variant callResult;

//Call Method1
callResult=helper.invoke("Primary::Method1","Test Value","Another String");

//Call Method2
va1.setInt(10);
va2.setInt(300);
callResult=helper.invoke("Primary::.Method2",va1,va2);
```

```
//Get Property1
callResult=helper.getProperty("Primary::Property1");

//Set Property2

Variant value=new Variant();
value.setString("Test Value:);

helper.setProperty2(value, "Primary::Property1","Key");

//Call Method 3 in the Secondary Interface
callresult=helper.invoke("Secondary::Method3","A String Value");

//Add A Property Listener for Property2
obj.addPropertyListener("Primary::Property2",listener);

//Remove a Property Listener
obj.removePropertyListener("Primary::Property2",listener);
```

Bulk Data Transfer mechanisms

The Java Gateway contains a number of data types that allow for the transfer of collections or large amounts of data in bulk to and from the server.

This section deals with what these objects are and how to use them.

Concepts

Below is a table of the bulk transfer objects (BTO) supplied in gateway package.

Types

Package com.esitechnology.eo

Interface	**Standard Implementation**	**EsiObjects Equivalent**	**Description**	**Collection?**
BTOList	List	ESI$List	A list of items	Yes
BTONVList	NVList	ESI$NVList	A list of items with Names	Yes
BTOTable	Table	ESI$Table	A multicolumn table	Yes
BTOText	Text	ESI$Text	A Large Text object	No

The List, NVList, and Table objects are collections into which elements can be inserted, removed and iterated.

Text objects deal with large amounts of string data. All of these objects can be created on the server and passed to the client. In other words, an ESI$List object created in EsiObjects can be returned to the client and used within the client as a List object. Conversely, if the client creates a List object and passes it into the server in a method parameter, the resultant object in the server is an ESI$List object.

The implementations of these objects are overrideable by creating a Subclass of BTOObjectFactory.

Streams

The Java Gateway will allow any of the following classes (or their descendants) to be passed to the EsiObjects server.

- java.io.InputStream
- java.io.OutputStream
- java.io.Writer
- java.io.Reader

On the EsiObjects server these objects will be represented as an ESI$TCPStream.

Transfer Dynamics

When a BTO object is passed as an argument (or used as a return value) the entire contents of that collection is transferred to the other side. In the case of a BTO object used as an argument of a call from the client to the server, the contents of the object will be transferred to the server in total, and at the conclusion of the call the new contents will be transferred back to the client.

Common Conventions

Index Basis

All structures have elements numbered from zero (0) to their Dimension minus one (1). Thus if you set the *Dimension* property of a List to 10, the indexes go from zero to nine.

Effects of Dimension

An attempt to set an element outside of the dimensioned bounds of a collection will result in an error. This means that all collections should be dimensioned prior to being filled. The only exception is when an append operation is being performed, which will cause the collection to grow.

Appending Items

Setting an item at position negative one (-1) will cause the item to be placed at the end of the collection and the dimension of the collection to be increased.

Common Error Handing

All BTO objects listed in the table above support a set of common error handling facilities. These facilities allow error information to be associated with the collection directly. These facilities might be used by methods that validate the contents of the collections.

java.io Object Support

The following classes (and their decendants) from java.io are understood by the gateway and will be transported to the server correctly. The server will be presented with an ESI$TCPStream object. This provides a mechanism whereby the server may read & write to the clients file system.

- java.io.InputStream
- java.io.OutputStream
- java.io.Writer
- java.io.Reader

Common Error Methods

The common error methods are supported by all BTO object that support "Data by Value". They provide a set of common facilities for associating error information with a BTO Object.

Details on these error methods are available in the Java Doc for com.esitechnology.eo.BTO

Bulk Transfer Objects (BTO)

BTOList

Overview

This object is used as a collection for a singleton list of items.

Please refer to the Java Doc for package **com.esitechnology.eo.BTOList** for details on the interface. For details on the implementation of List refer to the Java Doc for **com.esitechnology.eo.List**.

Within EsiObjects, this object type is represented as an ESI$List.

Example:

```
// Import the com.esitechnology.eo package
import com.esitechnology.eo.*;
// Define the List
List theList = new List();
// Set the size of the List via the Dimension property
theList.setDimension(12);
// Load the List with some data
 for (int item=0; item < theList.getDimension(); item++){
    Variant value = new Variant();
    value.setString("Item: " + item);
    theList.setElement(item, value);
}
```

```
// Append a few items into the List via the SetElement using -1
index
theList.setElement(-1, new Variant(“Extra Item 1”);
theList.setElement(-1, new Variant(“Extra Item 2”);
// Fill a JComboBox with items from the List
for (int item=0; item < theList.getDimension(); item++){
    value = theList.getElement(item);
    cbox.addItem(value.getString());
}
```

BTONvList

Overview

The NVList provides a list of named items. Each item on the list may be given a name. Names need not be unique. An index based on name is maintained which allows items to be retrieved by name. An NVList object can also be used like a Map.

Please refer to the Java Doc for package **com.esitechnology.eo.BTONvList** for details on the interface. For details on the implementation of List refer to the Java Doc for **com.esitechnology.eo.NvList**.

Within EsiObjects, this object type is represented as an ESI$NVList.

Example

In this example, the bridgeServer, object is a proxy object created from an EsiObjects class RGTest that contains various methods for testing the BTO objects.

```
private BridgeServer bridgeServer
bridgeServer=(BridgeServer)theGateway.simpleCreateObject("RGTest$BridgeSe
rver",
0,null);
.
.
.
private boolean NvListTest()  {

try{

    //NvList - Local Tests
    NvList nvList;
    nvList=new NvList();
    //Populating List
    for (int a=1;a<11;a++){
        String key;
        Variant value;
        key="Key"+a;
```

```
            value=new Variant();
            value.setInt(a);
            nvList.setPair(-1,key,value);
        }

        int size=nvList.getDimension();
        // UpdateStatus is a separate function to print output
        UpdateStatus("List Dimension = "+size);
        for (int a=0;a<size;a++){
            String key;
            Variant value;
            key=nvList.getName(a);
            value=nvList.getElement(a);
            UpdateStatus( "Key = " + key);
            UpdateStatus( "Value = "+value.getString());
        }

        //Remote Tests
        nvList.clearAll();
        Variant vaList;
        vaList = new Variant();
        vaList.setObject(nvList);
        // invoke the NVListText method on the RGTest object
        bridgeServer.NVListTest(vaList);
        size=nvList.getDimension();
        UpdateStatus("List Dimension = "+size);
        for (int a=0;a<size;a++){
            String key;
            Variant value;
            key=nvList.getName(a);
            value=nvList.getElement(a);
            UpdateStatus( "Key = " + key);
            UpdateStatus( "Value = "+value.getString());
        }
        //List as Return Value Test
        nvList=(NvList)bridgeServer.NVListRet().getObject();
        size=nvList.getDimension();
        UpdateStatus("List Dimension = "+size);
        for (int a=0;a<size;a++){
            String key;
            Variant value;
            key=nvList.getName(a);
            value=nvList.getElement(a);
            UpdateStatus( "Key = " + key);
            UpdateStatus( "Value = "+value.getString());
        }

    }
    catch(OutOfBoundsException exp){
        UpdateStatus(exp.toString());
        return false;
    }
    catch(NoConversion exp){
        UpdateStatus(exp.toString());
        return false;
    }
    catch(RequestFailed exp){
        UpdateStatus(exp.toString());
        return false;
    }

    return true;
}
```

BTOTable

Overview

The Table provides for a collection of data organized by rows & columns. The table is optimized for row based operations rather than column access. To be useful the table must always be dimensioned, although it is legal to specify zero for one of the two dimensions.

Please refer to the Java Doc for package **com.esitechnology.eo.BTOTable** for details on the interface. For details on the implementation of List refer to the Java Doc for **com.esitechnology.eo.Table**

Within EsiObjects, this object type is represented as an ESI$Table

Example

In this example, the bridgeServer, object is a proxy object created from an EsiObjects class RGTest that contains various methods for testing the BTO objects.

```
private BridgeServer bridgeServer
bridgeServer=(BridgeServer)theGateway.simpleCreateObject("RGTest$BridgeSe
rver",
0,null);
.
.
.
private boolean TableTest(){
// Test a BTO Table
try{
    //Table local Tests
    Table table;
    table=new Table();
    UpdateStatus("Dimensioning Table");
```

```
table.setDimension(0,4);
UpdateStatus("Populating Table");
Variant[] params;
params = new Variant[4];
for (int a=1;a<11;a++){
    params[0]=new Variant(a);
    params[1]=new Variant("Const string");
    params[2]=new Variant((float)3.1415);
    params[3]=new Variant(false);
    table.addRowV(params);
}
java.awt.Dimension size;
size=table.getDimensions();
for (int a=0;a<size.height;a++){
    StringBuffer buf;
    Variant value;
    buf=new StringBuffer();
    for (int b=0;b<size.width;b++){
        value=table.getCell(a,b);
        buf.append(value.getString());
        buf.append(" ");
    }
    UpdateStatus(buf.toString());
}
Variant vaTable;
vaTable=new Variant(table);
//Calling Server - Table as Argument
bridgeServer.TableTest(vaTable);
size=table.getDimensions();
for (int a=0;a<size.height;a++){
    StringBuffer buf;
    Variant value;
    buf=new StringBuffer();
    for (int b=0;b<size.width;b++){
        value=table.getCell(a,b);
        buf.append(value.getString());
        buf.append(" ");
    }
    UpdateStatus(buf.toString());  // function to print
}
//Calling Server - Table Return
table=(Table)bridgeServer.TableRet().getObject();
size=table.getDimensions();
UpdateStatus("Size = "+size.toString());
for (int a=0;a<size.height;a++){
    StringBuffer buf;
    Variant value;
    buf=new StringBuffer();
    for (int b=0;b<size.width;b++){
```

```
                value=table.getCell(a,b);
                buf.append(value.getString());
                buf.append(" ");
            }
            UpdateStatus(buf.toString());
        }

    }
    catch(OutOfBoundsException exp){
        UpdateStatus(exp.toString());
        return false;
    }
    catch(NoConversion exp){
        UpdateStatus(exp.toString());
        return false;
    }
    catch(RequestFailed exp){
        UpdateStatus(exp.toString());
        return false;
    }

    return true;
}
```

BTOText

Overview

The BTOText object supports very long block text. Given the restrictions of most M systems, large blocks of text must be store in a section of Blocks. The BTOText object is the client side representation of the ESI$Text object which supports unbounded text blocks. The operations on BTOText facilitate the breaking of the large text object into smaller chunks that can be stored in M.

Access is provide to the text in four manners:

1. As a single text string.
2. By line (As delimited by CR/LF.)
3. By a substring.
4. By text block of a given a specific size.

The TCP transport will automatically create this type of object on the server side when it receives text that is too large for the M system to handle.

Please refer to the Java Doc for package **com.esitechnology.eo.BTOText** for details on the interface.

For details on the implementation of List refer to the Java Doc for **com.esitechnology.eo.Text**.

Within EsiObjects, this object type is represented as an ESI$Text.

Example

In this example, the bridgeServer, object is a proxy object created from an EsiObjects class RGTest that contains various methods for testing the BTO objects.

```
private BridgeServer bridgeServer
bridgeServer=(BridgeServer)theGateway.simpleCreateObject("RGTest$BridgeSe
rver",
0,null);
.
.
.
private boolean TextTest(){
//Test BTO Text
try{
    //Local Tests
    Text text;
    text = new Text();
    //Simple SetText call
    text.setText("Set the text to a simple string");
    //UpdateStatus is a separate function to output a string
    UpdateStatus(text.getText());
    UpdateStatus("Clear");
    text.clear();
    UpdateStatus(text.getText());
    //Append Tests
    for (int a=1;a<100;a++){
        text.append("123456789012345678901234567890");
    }
    UpdateStatus(text.getText());
    UpdateStatus("Length = "+text.getLength());
    text.clear();
    //Line test
    for (int a=0;a<100;a++){
        text.append(a+"  123456789012345678901234567890\r\n");
    }
    UpdateStatus("Line Count = "+text.getLineCount());
    UpdateStatus("Line 13 = "+text.getLine(13));
    int blocks;
    blocks=text.getBlockCount(510);
    UpdateStatus("Length = "+text.getLength());
    UpdateStatus("Blocks = "+blocks+", 510 chars per block");
    for (int a=0;a<blocks;a++){
        UpdateStatus("Block "+a+" = "+text.getBlock(a,510));
    }
    //Text - Remote Tests
    Variant vaText;
```

```
        vaText = new Variant();
        vaText.setObject(text);
        //Calling TextTest
        bridgeServer.TextTest(vaText);
        //Viewing Returned text
        int lines;
        lines=text.getLineCount();
        UpdateStatus("Lines returned = "+lines);
        for (int a=0;a<lines;a++){
            UpdateStatus(text.getLine(a));
        }
        //Calling TextRet
        text=(Text)bridgeServer.TextRet().getObject();
        UpdateStatus("Viewing Returned text");
        lines=text.getLineCount();
        UpdateStatus("Lines returned = "+lines);
        for (int a=0;a<lines;a++){
            UpdateStatus(text.getLine(a));
        }

    }
    catch (StringIndexOutOfBoundsException exp)
    {
        UpdateStatus(exp.toString());
        return false;
    }
    catch(RequestFailed exp){
        UpdateStatus(exp.toString());
        return false;
    }

    return true;
}
```

Streams

The EsiObject TCPGateway supports the transport of stream objects by reference to the server. This provide a mechanism for stream based I/O between the Client and the EsiObjects Server. When the object is received by the EsiObjects Server it will be represented by and Instance of ESI$TCPStream.

The Gateway provides support for any object that extends one of the following classes:

- java.io.InputStream
- java.io.OutputStream
- java.io.Writer
- java.io.Reader

Event Processing

Overview

One of the most powerful features in EsiObjects is event processing. Objects can watch other objects for events or changes in state. The Java Gateway brings that same functionality to the client. Using the event processing capability of the gateway, a client can watch an object on the server for a specific event/state-change, or any event/state-change. When the event occurs, a specified method is invoked on the client (known as a “callback”.) Thus clients can register for and respond to events that occur on the server.

One of the most common uses for event processing is for keeping a client display of data current with what is on the server. Especially when more than one process is changing data on the server. The client can register for events that occur when data is changed. When the event is thrown, the callback method is invoked and that method could update the client display with the most recent data.

In EsiObjects we say that an object will “watch” another object for a specific event or category of events. The watched object will “throw” the event when it encounters it by placing the event on an event queue.

Below we describe how a client makes use of event processing using the gateway.

Process description

There are several steps that must be done for event processing to be implemented on the client.

- The programmer must write the callback method.
- The programmer must define an Event Listener. This is a datatype EOEventListener that contains the reference to the callback method for the event

In the client code the programmer must:

- Create an EOEventListener.
- Watch the object. This actually “registers” the fact that the client is going to watch a specified object for some type(s) of events. During the time that this watch is active, if the event occurs in the object, the specified callback method in the client is invoked.
- When the client no longer wants to watch the object, it should ignore the object. This deactivates the Watch.

Each of these steps is detailed below.

Create a Callback method

Callback methods are defined in the following format:

```
public void callbackmethodname(EOEvent evt)
```

The callback accepts one parameter of type EOEvent, which is defined in the gateway. EOEvent is a structure sent by the server when an event occurs. This structure contains information associated with the event: the name of the event, the object in which the event occurred, and other parameters containing information supplied by the server on the event.

Refer to the Java Doc for details on the EOEvent structure.

Typically, the callback will extract the data out of the EOEvent structure and use the data to take appropriate action.

Create a Listener

A Listener object is one that contains information for the callback method. Anytime a watch is invoked from the client, a Listener object must be associated with the watch. The constructor for the listener accepts a reference to the callback method associated with this listener.

Example

In the following example, a listener orgRelAddedListener is created. The callback method orgRelAddedEventOccurred is associated with this listener. Note that the listener has defined inline the implementation of the **eventOccurred()** method.

```
private EoEventListener orgRelAddedListener = null;
orgRelAddedListener = new EoEventListener() {
     public void eventOccurred(EOEvent evt) {
      orgRelAddedEventOccurred(evt); }}
```

Add Listener

Each event, property or unary relationship within a proxy has two event services generated for them: **add*ServiceName*Listener()** and **remove*ServiceName*Listener()**. (Where ***ServiceName*** is the name of an event, property or relationship.)

Both operations take a single argument: an EOEventListener. The EOEventListener provides the “callback” method to be invoked on the client whenever the specified event occurs in the server.

The **add*SeviceName*Listener()** establishes a watch on the server object represented by the proxy. When the specified *ServiceName* event occurs, the associated callback method is invoked.

Example

In the following example a proxy object (which is a Base$Set object) is being watched for *ElementAdded* events to occur. Whenever this event occurs in the server object, the callback associated with the orgRelAddedListener will be invoked.

```
employerSet.addElementAddedListener(orgRelAddedListener);
```

Handle Event

eventOccurred is invoked on each listener which in turn calls the callback associated with the event listener when it was created (see above.)

Remove Listener

The **remove*ServiceName*Listener()** method invoked on the proxy will remove the watch on the server object. This method takes a single argument: the EOEventListener associated with the watch to be removed.

Example

This example shows the removing of a watch on a set object for *ElementAdded* events.

```
employerSet.removeElementAddedListener(orgRelAddedListener);
```

What can be watched

- Properties
- Events
- Unary Relationships

To watch a relationship that is of cardinality “many”, you must first get the handle to the relationship object and then watch for events on that object.

The Event Queue

Each client connection to the server is allocated an event queue. Whenever an event occurs the event is place on the event queue for the client to pick up. There are two ways in which the event queue has its events pulled and sent to the client:

At the conclusion of a call to the server, the event queue for the connection is checked and all of the events are dispatched back to the client at that time.

The **dispatchEvents()** method of the Gateway may also be used to explicitly dispatch any events that might have been caused by the actions of others. This method essentially polls the server for any events on the queue and if any are found, they are dispatched to the client. Typically a Timer could be set up to execute this method at regular intervals to check for any events that have occurred.

Possible Complications

If the proxy object is not referenced, the event will be short circuited, and the listener will not be invoked. When the proxy object is *finalized*, an ignore will be issued to the server if there are any attached listeners.

What Hooks Are Generated

In the Proxy generated code, each property, unary relationship, and programmer-defined event has corresponding code generated.

Properties and Relationships

For example, for the Tutorial$Person class, the following was generated for the Address property. Relationships are generated in the same way.

```
import java.beans.PropertyChangeListener;
.
.
.
public void addAddressListener(PropertyChangeListener listener){
   javaProxy_addPropertyListener("Primary::Address",listener);
}
public void removeAddressListener(PropertyChangeListener
            listener){
   javaProxy_removePropertyListener("Primary::Address",listener);
}
```

Events

For the Tutorial$Database class, the programmer-defined event *Deleted* had the following code generated.

```
public void addDeletedListener(EoEventListener listener){
   javaProxy_addEventListener("Primary::Deleted",listener);
}
public void removeDeletedListener(EoEventListener listener){
   javaProxy_removeEventListener("Primary::Deleted",listener);
}
```

Polling for Events

When your application is sitting idle, you may want to check the server to see if there are any events that have occurred. The Gateway provides the method **dispatchEvents()** that will dispatch any events that may be enqueued on the server.

Using dispatchEvents()

The usage of **dispatchEvents()** is rather simple. It simply needs to be invoked. It is suggested that this be done in either the applications idle time processing or by using a timer.

Controlling Event Dispatch

There are several mechanisms for controlling event processing (all available in the Gateway):

- Event processing can be enabled/disabled using the **setEventDispatchEnabled()** method.
- The number of events outstanding can be checked using the **getNumberOfEventsPresent()** method.
- You can determine if event processing is enabled by using the **getEventDispatchEnabled()** method.

Advanced Usage

The following sections deal with using advanced features of the Gateway. Apart from invoking services on Proxy objects, you can also deal with the Gateway directly to invoke services on objects, make use of event processing etc. In fact you must use the Gateway directly in order to Create and Destroy objects.

In addition to services provided by the Gateway, the Gateway can have its functionality extended by providing handlers for the server to use. An example would be providing the server with a Handle to an output window so that server output messages could be directed to that object.

Direct Gateway Functions

The Java Gateway includes services that allow for direct invocations on the server. For example, an object on the server can have a method invoked on it without a proxy.

The following sections summarize the functions provided by the Gateway.

General Purpose Services

The Gateway provides a couple of calls that are invoked on the Gateway itself and provide general services.

isReady() – Returns a boolean indicating if the Gateway is connected.

lookupObject() – Service to find objects on the server (see above.)

Object Services

Object services are those methods, properties, and relationships implemented by an object. In General, you use a generated Proxy object to invoke these services on a server object. However, the Gateway does provide for invoking services of an object directly, via an API.

General Conventions

To invoke a method, property, or relationship on a server object, you need the object id, the name of the service, and an array of parameters if parameters are required for the call. Additionally, you can specify certain flags on the call that affect how the service is delivered to the object.

The Call Flags

The call flags are used on a number of object service call points and alter the basic behavior to the call. In general these flags suppress the generation of errors on the server. The flags may be comibined using a Boolean Or operator (|). If now special handling is desired then the falg MODEIFIERFG_NONE should be passed.

MODIFIERFG_NONE	No special handling is desired.
MODIFIERFG_EXISTENCE	Causes the invocation to be ignored if the target object does not exist. Normally an error results if the target object does not exist
MODIFIERFG_KNOWS	Causes the invocation to be ignored if the target object does understand the message. Normally an error results if the target object does support the operation
MODIFIERFG_FILTER	Causes unknown parameters to be ignored (Filtered).

The Parameters Array

The Gateway provides a data type called Parameter. See the Java Docs for information on this data type. When passing parameters on the calls below, an array of these parameters is what is used. This provides a mechanism whereby named parameters may be passed to an object.

Specific Services

invoke

Invokes a method on a specified object. The modifier flags are set in the third parameter, and any parameters are placed in the parameter array.

```
public Variant invoke(java.lang.String object,
                      java.lang.String method,
                      int modifierFlags,
                      Parameter[] paramArgs)
               throws RequestFailed
```

propertyGet

Invokes the **Get** property accessor method on a specified object. The modifier flags are set in the third parameter, and any parameters are placed in the parameter array.

```
public Variant propertyGet(java.lang.String object,
                           java.lang.String property,
                           int modifierFlags,
                           Parameter[] paramArgs)
                    throws RequestFailed
```

propertySet

Invokes the **Assign** property accessor method on a specified object. The value to assign to the property is specified in the first parameter. The modifier flags are set in the fourth parameter, and any parameters are placed in the parameter array.

```
public boolean propertySet(Variant newValue,
                           java.lang.String object,
                           java.lang.String property,
                           int modifierFlags,
                           Parameter[] paramArgs)
                    throws RequestFailed
```

propertyKill

Invokes the **Kill** property accessor method on a specified object. The modifier flags are set in the third parameter, and any parameters are placed in the parameter array.

```
public boolean propertyKill(java.lang.String object,
                            java.lang.String property,
                            int modifierFlags,
                            Parameter[] paramArgs)
                     throws RequestFailed
```

propertyDataFn

Invokes the **Data** property accessor method on a specified object. The modifier flags are set in the third parameter, and any parameters are placed in the parameter array.

```
public Variant propertyDataFn(java.lang.String object,
                              java.lang.String property,
                              int modifierFlags,
                              Parameter[] paramArgs)
                       throws RequestFailed
```

propertyGetFn

Similar to the **propertyGet**() above, this service Invokes the **Get** property accessor method on a specified object. The difference is that a default value is specified if the property is not set. The modifier flags are set in the fourth parameter, and any parameters are placed in the parameter array.

```
public Variant propertyGetFn(Variant defaultVal,
                             java.lang.String object,
                             java.lang.String property,
                             int modifierFlags,
                             Parameter[] paramArgs)
                      throws RequestFailed
```

propertyOrderFn

Invokes the **Order** property accessor method on a specified object. The modifier flags are set in the fourth parameter, and any parameters are placed in the parameter array. The first parameter specifies whether the order should go forward (1) or reverse (-1).

```
public Variant propertyOrderFn(Variant direction,
                               java.lang.String object,
                               java.lang.String property,
                               int modifierFlags,
                               Parameter[] paramArgs)
                        throws RequestFailed
```

propertyQueryFn

Invokes the **Query** property accessor method on a specified object. The modifier flags are set in the third parameter, and any parameters are placed in the parameter array.

```
public Variant propertyQueryFn(java.lang.String object,
                               java.lang.String property,
                               int modifierFlags,
                               Parameter[] paramArgs)
                        throws RequestFailed
```

propertyNormalizeFn

Invokes the **$Normalize** property accessor method on a specified object. The modifier flags are set in the fourth parameter, and any parameters are placed in the parameter array. The first parameter is the value being normalized.

```
public Variant propertyNormalizeFn(Variant value,
                                   java.lang.String object,
                                   java.lang.String property,
                                   int modifierFlags,
                                   Parameter[] paramArgs)
                            throws RequestFailed
```

propertyValidateFn

Invokes the **$Valid** property accessor method on a specified object. The modifier flags are set in the fourth parameter, and any parameters are placed in the parameter array. The first parameter is the value being validated.

```
public Variant propertyValidateFn(Variant value,
                                  java.lang.String object,
                                  java.lang.String property,
                                  int modifierFlags,
                                  Parameter[] paramArgs)
                           throws RequestFailed
```

Life Cycle Services

The gateway provides three services for creating and destroying objects.

createObject()

Creates an object on the server allowing the programmer to use most of the functionality available to create an object. Positional and named parameters can be used, along with property assignments.

```
public CoreObject createObject(java.lang.String className,
                               int createFlags,
                               BTONvList createKeywords,
                               BTOList params,
                               BTONvList namedParams,
                               BTONvList properties)
                        throws RequestFailed
```

simpleCreateObject()

Creates an object using a simplified call. Positional parameters only.

```
public CoreObject simpleCreateObject(java.lang.String className,
                                     int createFlags,
                                     Variant[] params)
                              throws RequestFailed
```

destroyObject

Destroys a specified object on the server.

Destroy an Object, given the objects OID as a string:

```
public void destroyObject(java.lang.String object)
                             throws RequestFailed
```

Destroy an Object, given a IntefaceBase derived class:

```
public void destroyObject(CoreObject object)
                         throws RequestFailed
```

Destroy an Object, given a Variant with the object in it:

```
public void destroyObject(Variant object)
                             throws RequestFailed
```

Event Processing

The following services deal with event processing.

dispatchEvents

This method of the Gateway may be used to explicitly dispatch any events that might have been caused by the actions of other objects. This method essentially polls the server for any events on the queue and if any are found, they are dispatched to the client. Typically a Timer could be set up to execute this method at regular intervals to check for any events that have occurred.

getEventDispatchEnabled

You can determine if event processing is enabled by using the **getEventDispatchEnabled()** method. This method returns a Boolean.

setEventDispatchEnabled

Event processing can be enabled/disabled using the **setEventDispatchEnabled()** method. This provide a control mechanism to avoid dispatch during certain calls or while a specific process is being exceuted.

getNumberOfEventsPresent

The number of events outstanding can be checked using the **getNumberOfEventsPresent()** method. This may be helpful in the management of the event dispatching.

watch

This service is used only by proxy object to take out watches on the server object. It should not be invoked directly.

ignore

This service is used only by proxy object to remove watches taken on the server object. It should not be invoked directly.

Service Hooks

The Gateway provides a number of service hook points that allow the developer to customize the gateway’s functionality. In all cases pass a null value to the service hook will restore the default.

Handler	**Description**
setOutputHandler()	Establish the object that will handle Server output messages
setStatusHandlerFactory()	Establish the Factory used for Status request message handlers
setClientServicesImplementation()	Establishes the ClientServices
setBTOObjectFactory()	Establishes the BTOObjectFactory that the Gateway will uses to create BTO objects
setDebugger()	Establishes what debugger should be used by the gateway (if any)

Object Debugging

The Gateway provides a single method relating to object debugging, called getDebuggingServices. This service will get the DebuggingServices object associated with the gateway. For security reasons some implementations of the gateway may return null for this service.

getDebuggingServices

Invoking this method on the Gateway will return an instance of the DebuggingServices interface. As mentioned above, some implementations may return NULL on this call. The Java Docs reference contains more information on the DebuggingServices interface.

This interface contains services such as ObjectInfo, ObjectXecute, ObjectEval, among others. These calls are not stack related – in other words, the execution of the service happens in a stateless manner. State from previous invocations is not preserved.

Note: These services may not be available on all gateway implementations, and should only be used for development, never for deployment.

Handlers

Handlers are interfaces that may be hooked to the gateway to handle specific types of server requests. In general they allow the application to implement extensions to the Gateway’s behavior.

Client Services

This handler allows for common user interface services from the server. These services are: messageBox, messageBeep, reportError, reportWarning.

Interface to Implement: ClientServices

Attachment call: setClientServicesImplementation

Output

Common handler for outputting text messages from the server. An example implementation is the Outputwindow in the EsiObjects development client.

Interface to Implement: OutputHandler

Attachment call: setOutputHandler

Status

Creates a StatusHandler instances for Server wait requested. An example is the Status windows in the EsiObjects Development client.

Interface to Implement: StatusHandlerFactory

Attachment call: setStatusHandlerFactory

BTO Object Factory

Allows the implementer to override the default BTO Objects that are created by the gateway. For example this make it possible for the implementer to implement their own version of BTOTable and have it returned to them by the Gateway.

Interface to Implement: BTOObjectFactory

Attachment call: setBTOObjectFactory

Debugger

Service will handle debugging messages from the server.

Interface to Implement: DebugSink

Attachment call: setDebugger