



Using the Java Gateway

Version 2019.1
2019-03-19

Using the Java Gateway

InterSystems IRIS Data Platform Version 2019.1 2019-03-19

Copyright © 2019 InterSystems Corporation

All rights reserved.



InterSystems, InterSystems Caché, InterSystems Ensemble, InterSystems HealthShare, HealthShare, InterSystems TrakCare, TrakCare, InterSystems DeepSee, and DeepSee are registered trademarks of InterSystems Corporation.



InterSystems IRIS Data Platform, InterSystems IRIS, InterSystems iKnow, Zen, and Caché Server Pages are trademarks of InterSystems Corporation.

All other brand or product names used herein are trademarks or registered trademarks of their respective companies or organizations.

This document contains trade secret and confidential information which is the property of InterSystems Corporation, One Memorial Drive, Cambridge, MA 02142, or its affiliates, and is furnished for the sole purpose of the operation and maintenance of the products of InterSystems Corporation. No part of this publication is to be used for any other purpose, and this publication is not to be reproduced, copied, disclosed, transmitted, stored in a retrieval system or translated into any human or computer language, in any form, by any means, in whole or in part, without the express prior written consent of InterSystems Corporation.

The copying, use and disposition of this document and the software programs described herein is prohibited except to the limited extent set forth in the standard software license agreement(s) of InterSystems Corporation covering such programs and related documentation. InterSystems Corporation makes no representations and warranties concerning such software programs other than those set forth in such standard software license agreement(s). In addition, the liability of InterSystems Corporation for any losses or damages relating to or arising out of the use of such software programs is limited in the manner set forth in such standard software license agreement(s).

THE FOREGOING IS A GENERAL SUMMARY OF THE RESTRICTIONS AND LIMITATIONS IMPOSED BY INTERSYSTEMS CORPORATION ON THE USE OF, AND LIABILITY ARISING FROM, ITS COMPUTER SOFTWARE. FOR COMPLETE INFORMATION REFERENCE SHOULD BE MADE TO THE STANDARD SOFTWARE LICENSE AGREEMENT(S) OF INTERSYSTEMS CORPORATION, COPIES OF WHICH WILL BE MADE AVAILABLE UPON REQUEST.

InterSystems Corporation disclaims responsibility for errors which may appear in this document, and it reserves the right, in its sole discretion and without notice, to make substitutions and modifications in the products and practices described in this document.

For Support questions about any InterSystems products, contact:

InterSystems Worldwide Response Center (WRC)

Tel: +1-617-621-0700

Tel: +44 (0) 844 854 2917

Email: support@InterSystems.com

Table of Contents

About This Book	1
1 Introduction to the Java Gateway	3
1.1 Prerequisites	3
1.2 Starting and Stopping the Gateway	4
1.3 Connecting and Disconnecting	4
1.4 Java Gateway Modes	5
1.4.1 Proxy Object Mode	5
1.4.2 Stateless Service Mode	8
2 Using Dynamic Object Gateways	11
2.1 Introducing Dynamic Gateways	11
2.2 Creating and Using a Dynamic Object Gateway	12
2.3 Array Access	13
2.4 Gateway Reentrancy	14
2.5 The Java Fruit Class	15
3 Using the Java Gateway in a Production	17
3.1 Adding the Java Gateway Business Service	17
3.2 Settings for the Java Gateway Business Service	18
3.3 Calling Business Service Methods	19
3.3.1 StartGateway() Method	20
3.3.2 ConnectGateway() Method	20
3.3.3 StopGateway() Method	20
3.4 Creating a Business Operation	20
3.5 Calling API Methods	21
3.5.1 %Connect() Method	21
3.5.2 %Disconnect() Method	21
3.5.3 %Shutdown() Method	21
3.5.4 %Import() Method	22
3.5.5 %ExpressImport() Method	23
3.5.6 %ClassForName() Method	23
3.5.7 %GetAllClasses() Method	23
3.6 Using the Command Prompt	23
3.7 Using the Java Gateway Wizard	24
3.8 Error Checking	24
3.9 Troubleshooting	25
4 Sample Code	27
4.1 Setting Up Java Gateway Examples	27
4.2 Running Plain Java Examples	28
4.3 Running JDBC Examples	28
4.4 Running EJB Gateway Examples	29
4.5 Running JMS Gateway Examples	29
4.5.1 JMS Point-to-Point (P2P) Example	29
4.5.2 JMS Publish/Subscribe (Pub/Sub) Example	30
4.5.3 Java Naming and Directory Interface (JNDI) Example	30
4.6 Stateless Service Mode Example	31
5 Mapping Specification	33

5.1 Package and Class Names	33
5.2 Primitives	33
5.3 Date and Time	34
5.4 Properties	35
5.5 Methods	35
5.5.1 Overloaded Methods	35
5.5.2 Method Names	36
5.5.3 Static Methods	36
5.6 Constructors	36
5.7 Constants	36
5.8 Java Classes	37
5.8.1 Java Object Superclass (java.lang.Object)	37
5.8.2 Java Arrays	37
5.8.3 Java Collections Framework	38
5.8.4 Recasting	39
5.8.5 Java Standard Output Redirection	39
5.9 Restrictions	39

List of Figures

Figure 1–1: Connecting to a Java Gateway Worker Thread	5
Figure 1–2: Java Gateway Operational Model	7
Figure 1–3: Importing Java Classes	8

About This Book

This book explains how to enable easy interoperability between InterSystems IRIS™ and Java components. The Java Gateway can instantiate an external Java object and manipulate it as if it were a native object within InterSystems IRIS.

This book contains the following chapters:

- [Introduction to the Java Gateway](#)
- [Using Dynamic Gateways](#)
- [Using the Java Gateway in a Production](#)
- [Sample Code](#)
- [Mapping Specification](#)

For a detailed outline, see the [table of contents](#).

The following books provide related information:

- *[Best Practices for Creating Productions](#)* describes best practices for organizing and developing productions.
- *[Developing Productions](#)* explains how to perform the development tasks related to creating a production.
- *[Configuring Productions](#)* describes how to configure the settings for productions, business hosts, and adapters. It provides details on settings not discussed in this book.

1

Introduction to the Java Gateway

This chapter introduces the Java Gateway, which provides an easy way for InterSystems IRIS™ to interoperate with Java components. It discusses the following topics:

- [Prerequisites](#)
- [Starting and Stopping the Gateway](#)
- [Connecting and Disconnecting](#)
- [Java Gateway Modes](#)

1.1 Prerequisites

The Java Gateway server runs within a JVM, which can be on the same machine as InterSystems IRIS or on a different machine. Complete the following setup steps on the machine on which the Java Gateway will run:

1. Install the Java Runtime Environment (for example, JRE 1.8.0_67).
2. Make a note of the location of the installation directory for JRE. This is the directory that *contains* the subdirectories `bin` and `lib`.

This is the value that you would use for `JAVA_HOME` environment variable. For example: `c:\Program Files\Java\jre8`

You use this information later when you configure your production.

3. Also make a note of the Java version. If you are uncertain about the Java version, open a DOS window, go to the `bin` subdirectory of your Java installation, and enter the following command:

```
java.exe -version
```

You should receive output like the following, depending on your platform:

```
java version "1.8.0_67"  
Java(TM) SE Runtime Environment (build 1.8.0_67-b24)  
Java HotSpot(TM) 64-Bit Server VM (build 23.19-b22, mixed mode)
```

It is not necessary to set any environment variables. To access the JVM, InterSystems IRIS uses information contained in the production.

1.2 Starting and Stopping the Gateway

Start

Before you can use the Java Gateway, you must use some mechanism to start the Java Gateway server and tell InterSystems IRIS the name of the host on which the Java Gateway server is running. You can start the Java Gateway server in one of the following ways:

- *Automatically*, by adding a Java Gateway business service to the production. The Java Gateway server starts when the production starts.
- *Manually*, by calling the business service **StartGateway()** method.
- *Manually*, by entering a command at the Terminal command prompt.

Note: If you are using JMS on the Java side, you must start the Java Gateway server in JMS mode (the command is slightly different).

Stop

Once started, the Java Gateway server runs until it is explicitly shut down. You can stop the Java Gateway server in one of the following ways:

- *Automatically*, by adding a Java Gateway business service to the production. The Java Gateway server stops when the production stops.
- *Manually*, by calling the **StopGateway()** method of the business service.
- *Manually*, by calling the **%Shutdown()** method of the Java Gateway API (EnsLib.JavaGateway.JavaGateway).

When using the Java Gateway with a production, it is a good practice to have the production start the Java Gateway server at production startup, and stop it at production shutdown. This happens automatically if you add a Java Gateway business service to the production.

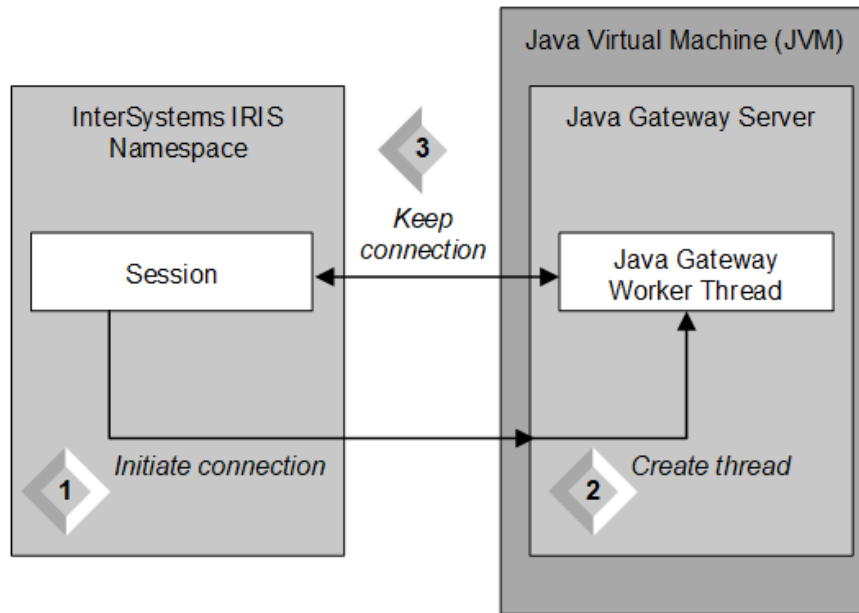
Note: If you make changes to your Java classes and want them available to the Java Gateway, you can stop and then restart the Java Gateway using any of these methods.

1.3 Connecting and Disconnecting

Once the Java Gateway server is running, each InterSystems IRIS session that needs to invoke Java class methods must create its own connection to the Java Gateway server. You can connect InterSystems IRIS with the Java Gateway server by calling the **ConnectGateway** method of the business service, or by calling the Java Gateway API **%Connect()** method.

The **Connect** command sets off the chain of sequential events (1), (2), and (3) shown in the following *Connecting to a Java Gateway Worker Thread* diagram:

Figure 1–1: Connecting to a Java Gateway Worker Thread



1. ObjectScript code sends a connection request.
2. Upon receiving the request, the Java Gateway server starts a worker thread in which the Java class methods subsequently run.
3. The connection between this Java Gateway worker thread and the corresponding InterSystems IRIS session remains established until it is explicitly disconnected.

ObjectScript code that establishes a worker thread must explicitly disconnect before exiting. Otherwise, the assigned port for the connection stays “in use” and is unavailable for use in other connections. ObjectScript code can disconnect its thread by calling the Java Gateway API `%Disconnect()` method.

1.4 Java Gateway Modes

Java Gateway has two main modes of manipulation of Java objects:

- **Proxy Object Mode** — Allows you to statefully manipulate Java Objects from within InterSystems IRIS.
- **Stateless Service Mode** — Allows you to make simple calls to Java methods and return the results from within InterSystems IRIS.

1.4.1 Proxy Object Mode

Proxy Object Mode provides an easy way for InterSystems IRIS to interoperate with Java components. It allows Java Gateway to instantiate an external Java object and manipulate it as if it were a native object within InterSystems IRIS.

Note: Static and Dynamic Proxy Objects

There are currently two kinds of proxy objects:

- *Static proxies* (described in this section) are standard ObjectScript classes, which must be individually generated and compiled before they can be used. If a new version of the Java class changes the interface in any way, the corresponding proxy class must be regenerated and recompiled to match it.
- *Dynamic proxies* (described in the next chapter, [Using Dynamic Gateways](#)) are created at runtime by introspecting the current version of the Java class. They are not projected and stored as standard ObjectScript classes; they are simply created and used as needed by your application, and always reflect the current version of the Java class.

In use, dynamic proxies look and work exactly like static proxies. The only difference is how they are created. Static proxies have some useful features, and will continue to be supported, but dynamic proxies are recommended for most new development.

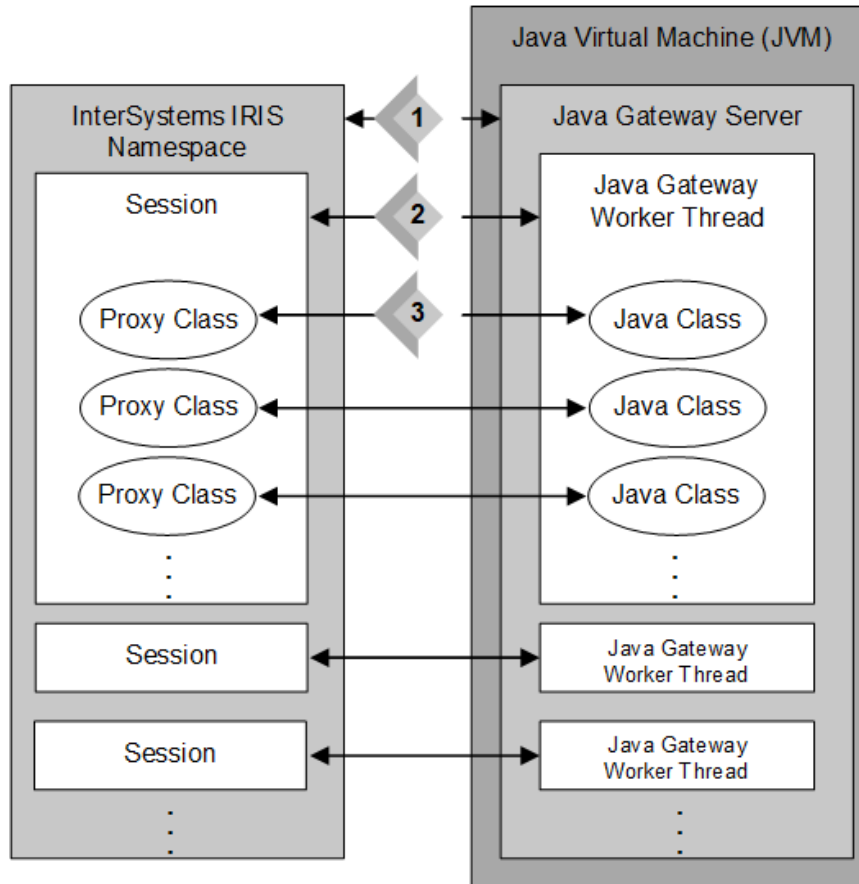
1.4.1.1 Proxy Object Mode Architecture

The external Java object is represented within InterSystems IRIS by a “wrapper” or “proxy” class. The proxy object appears and behaves just like any other InterSystems IRIS object, but it has the capability to issue method calls out to a Java virtual machine (JVM), either locally or remotely. Any method call on the proxy object triggers the appropriate class method inside the JVM.

You can use the Java Gateway to create proxy InterSystems IRIS classes for custom Java components. You can also create proxy mappings to existing Java interface specifications, such as the Java Database Connection (JDBC), Java Message Service (JMS), Enterprise Java Beans (EJB), Java Connector Architecture (JCA), etc. InterSystems IRIS can use this mapping to work with any implementation that is compliant with one of these specifications.

When generating proxies from large Java libraries and frameworks, the best approach is to build a small wrapper class that exposes just the functionality you want and then create a proxy for this wrapper. This makes the API between InterSystems IRIS and Java very clean and eliminates many potential issues dealing with how to map more esoteric features to a proxy object.

The following diagram provides a conceptual view of InterSystems IRIS and the Java Gateway at runtime while using Proxy Object Mode.

Figure 1–2: Java Gateway Operational Model

The Java Gateway server runs in the JVM. InterSystems IRIS and the JVM may be running on the same machine or on different machines. The items (1), (2), and (3) in the preceding diagram represent the relationships established by the commands that set up this operational model:

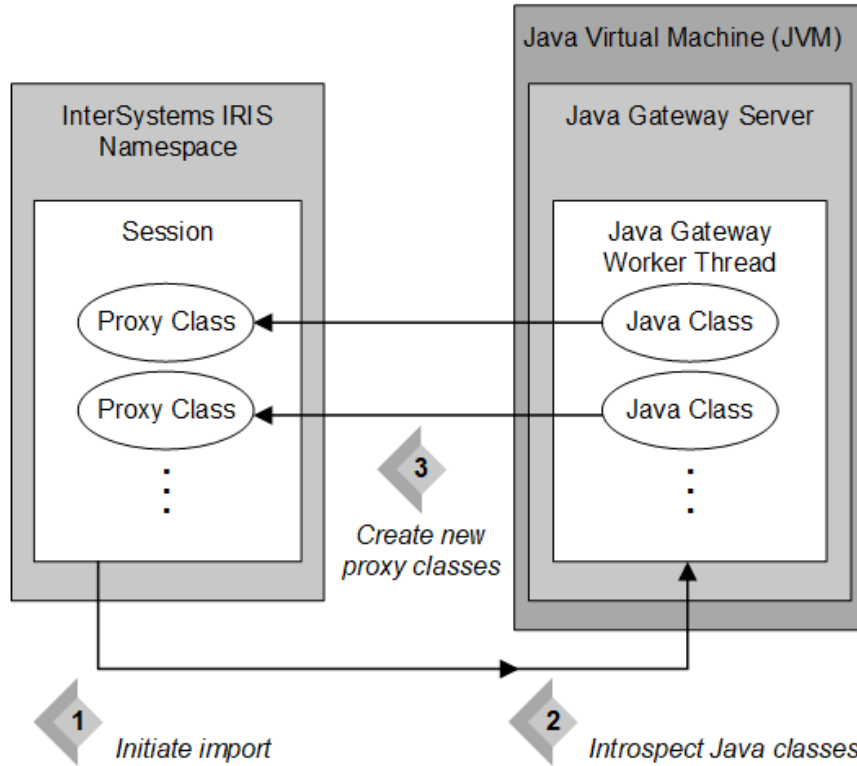
1. [Start](#)
2. [Connect](#)
3. [Import](#)

Later sections in this chapter explain how these commands work to set up InterSystems IRIS proxy classes for Java code, as well as how the proxies work once these relationships are set up; see “[Proxy Call Sequence](#)” for details.

1.4.1.2 Importing Java Classes

The Java Gateway API `%Import()` method sets off the chain of sequential events (1), (2), and (3) shown in the following diagram:

Figure 1–3: Importing Java Classes



1. The InterSystems IRIS session sends an import request.
2. Upon receiving the request, the Java Gateway worker thread introspects the indicated Java packages and classes.
3. If it finds any Java classes that are new or changed, or that have no proxy classes on the InterSystems IRIS side, the thread generates new proxy classes for them.

Important: The Java Gateway import only imports classes, methods, and fields marked as `public`.

1.4.1.3 Proxy Call Sequence

A call to any InterSystems IRIS proxy method initiates the following sequence of events:

1. All InterSystems IRIS proxy parameters are written into the output buffer.
2. A message containing the method name and parameters is sent to the Java Gateway worker thread.
3. The Java Gateway worker thread consumes the message, reads the parameters, finds the appropriate method or constructor call, and invokes it using Java reflection. If the given method is an overloaded method, the gateway uses a method overload algorithm to find the right Java method version. For details, see “[Overloaded Methods](#)” in the chapter “Mapping Specification.”
4. The results of the method invocation (if any) are sent back to the InterSystems IRIS side over the same connection.
5. The InterSystems IRIS proxy processes any return values and the method call returns.

1.4.2 Stateless Service Mode

The Stateless Service Mode allows simple and efficient calls out to a particular Java service. A Java service is any implementation of the `com.intersystems.gateway.Service` interface. Only the following method needs to be implemented:

```
public byte[] execute(byte[] args) throws Throwable;
```

This method takes a byte array, performs whatever service it needs to do, and produces a `byte[]` result. In order to invoke the above Java service method from InterSystems IRIS, call the following `%Net.Remote.Gateway` method:

```
Method %ServiceRequest(serviceName As %String, arguments As %String, ByRef response As %String) As %Status
```

Where `serviceName` is the name of the implementing Java service class, `arguments` corresponds to the Java service `args` and

`response` corresponds to the Java service `result`. `arguments` and `result` are represented as `%Strings` on the InterSystems IRIS, and byte arrays on the Java side, meaning any values serialized as such will be accepted by the underlying engine.

The following static method to `%Net.Remote.Gateway` directly allows invocation of an external Java service:

```
ClassMethod %RemoteService(host As %String, port As %Integer, serviceName As %String, arguments As %String, additionalClassPaths As %ListOfDataTypes = "") As %String;
```

Note: The implementation of Java service should never include a callback to InterSystems IRIS as this newly added component is not designed to be reentrant.

For a simple implementation using GSON please see [Stateless Service Mode Example](#).

2

Using Dynamic Object Gateways

This chapter describes how to implement Object Gateways using *dynamic proxy objects*. In previous releases, *static proxies* had to be generated separately, compiled, and stored in the class library just like any other ObjectScript class (see “[Importing Java Classes](#)” in the previous chapter).

Dynamic proxy objects do not require a separate import process before the business operation can start. The business operation simply creates and uses them as needed during runtime.

- [Introducing Dynamic Gateways](#) — provides an overview of dynamic proxies.
- [Creating and Using a Dynamic Object Gateway](#) — describes how to use dynamic proxies and gives specific usage examples.
- [Array Access](#) — describes how Java arrays are implemented as dynamic proxies.
- [Gateway Reentrancy](#) — describes how to ObjectScript and Java code can run safely using the same connection and context.
- [Example Classes](#) — contains detailed information on the classes used as examples in this chapter.

2.1 Introducing Dynamic Gateways

All versions of the Object Gateway can instantiate an external Java object and manipulate it as if it were a native InterSystems IRIS object. This is accomplished by creating a proxy object that provides all of the same methods and properties as the corresponding Java object. Any call on the proxy object triggers the corresponding method or property of the Java object. There are currently two kinds of proxy objects:

- *Static proxies* are standard ObjectScript classes, which must be individually generated and compiled before they can be used. If a new version of the Java class changes the interface in any way, the corresponding proxy class must be regenerated and recompiled to match it.
- *Dynamic proxies* are created at runtime by introspecting the current version of the Java class. All dynamic proxies are instances of `%Net.Remote.Object`. Although each Object instance appears to have the same methods and properties as the corresponding Java object, it actually uses an internal map of the interface to simulate these features.

In use, static and dynamic proxies are identical except for the way they are instantiated. For example:

```
// Static proxy: call %New() on a pregenerated class
set newFruit = ##class(Fruit).%New()

// Dynamic proxy: create an Object instance and specify the class to introspect
set newFruit = ##class(%Net.Remote.Object).%New(gateway,"Fruit")
```

When the code is used in a business production, the only noticeable difference is that dynamic gateways eliminate the import process.

2.2 Creating and Using a Dynamic Object Gateway

This section describes the complete procedure for creating and running an Object Gateway that uses dynamic proxies:

- [Get the server definition and test the server](#) — the server definition is required for methods that monitor and control the server.
- [Define the location of your Java classes and create a Gateway connection](#) — connection information must include the path to each .class or .jar file you intend to use.
- [Create a proxy and call some methods](#) — after they've been created, static and dynamic proxies work the same way.
- [Disconnect and shut down](#) — the server will continue to run until you explicitly stop it.

Get the server definition and make sure the server is running

The `Service` class provides methods to start, stop, and test the server. Most of these methods require the server definition contained in an `ObjectGateway` object. In the following example, `Service.OpenGateway()` returns an `ObjectGateway` instance defining a server named `JavaGate`. `Service.IsGatewayRunning()` uses the information to test the current state of the server, and `Service.StartGateway()` can start it if necessary.

```
try {
// Get a ObjectGateway instance that defines the JavaGate server
set st = ##class(%Net.Remote.Service).OpenGateway("JavaGate",.OG)
write !, "Using "_OG.Name_" server definition"

// Test to make sure the server is running. If not, start it
if ('##class(%Net.Remote.Service).IsGatewayRunning(OG.Server,OG.Port,,.status)) {
// Instantiate the server on the host machine
set status = ##class(%Net.Remote.Service).StartGateway(OG.Name)
}
}
catch ex {
write !,$system.OBJ.DisplayError(ex.AsStatus())
}
```

Note: Exception Handling

Your ObjectScript code should always use `try/catch` for Object Gateway error handling. When the class running on the host machine encounters an exception, the dynamic proxy will throw `%Net.Remote.Exception`. In addition to standard status details, the `Exception` object contains call stack information, which is returned by calling method `Exception.StackAsArray(.array)`. The information may be different between Java and .NET.

Define the location of your Java classes and create a Gateway connection

Each dynamic proxy will access the corresponding Java object through a `Gateway` object connected to the server. The `Gateway` object needs a path to each .class or .jar file containing a class you want to use. In this example, the location of the .jar file containing class `Fruit` is specified in the last argument to `Gateway.%Connect()`.

```
// define the location of the file containing the Fruit class
set fruitPath = ##class(%ListOfDataTypes).%New()
do fruitPath.Insert("C:\Dev\Fruit.jar")

// Connect a Gateway instance to server JavaGate on the host machine
set GW = ##class(%Net.Remote.Gateway).%New()
set st = GW.%Connect(OG.Server, OG.Port, "USER",,fruitPath)
```

This example only adds one path, but you can insert a separate path for each .class or .jar file you want to use.

Create a dynamic proxy for Java class Fruit and call some methods

In this example, a new instance of Object is created, specifying Fruit as the Java class to introspect. The resulting dynamic proxy will communicate with the Java object through Gateway instance *GW* (created in the previous example). Once the dynamic proxy is instantiated, it works just like a static proxy.

```
// Use GW connection to create a proxy for the Fruit class
set proxyFruit = ##class(%Net.Remote.Object).%New(GW,"Fruit")

write !,proxyFruit.id()
write !,"Current fruit preference: ",!, "  "_proxyFruit.getFruit()
do proxyFruit.setFruit("Jujube")
write !,"Fruit preference has been reset: ",!, "  "_proxyFruit.getFruit()
```

Prints:

```
This Java class displays your favorite fruit.
Current fruit preference: UglyFruit
Fruit preference has been reset: Jujube
```

Disconnect and shut down the server

The connection remains open and the server continues to run until you explicitly disconnect and shut down. In this example, the Gateway.**%Disconnect()** method breaks the connection, but the server won't release the port it uses until the call to Service.**StopGatewayObject()** stops it.

```
// Disconnect from the server
set st = GW.%Disconnect()

// Stop the server
set st = ##class(%Net.Remote.Service).StopGatewayObject(OG)
```

2.3 Array Access

When a property is declared as an array, it is projected to the proxy as an instance of %Net.Remote.Object. To create a new array, call the Object.**%New()** method and use array class name syntax in place of the regular class name. Arguments are used as the initial values for the array. Here are some examples:

Array initialization

Array can be initialized by rank or elements:

```
// by rank
set newArray = ##class(%Net.Remote.Object).%New(gateway,"String[2]")

// by elements
set newArray = ##class(%Net.Remote.Object).%New(gateway,"String[]","A","B","C")
```

When rank and elements are both specified, rank must match element count.

```
// both rank and elements specified
set newArray = ##class(%Net.Remote.Object).%New(gateway,"int[3]",11,22,33)
```

Getting and setting array elements

Besides supporting all the normal properties and methods available on an array object, Dynamic Gateway supports two special methods for accessing array elements:

- `%get(index)` — `object.%get(index)` is translated to `object[index]`
- `%set(index,value)` — `object.%set(index,value)` is translated to `object[index] = value`.

For example:

```

set max = 5
set arrSize = "String[_(max+1)_]" // "String[6]"
set arr = ##class(%Net.Remote.Object).%New(gateway,arrSize)

// Populate the array using %set()
for idx=0:1:max {
    set val = $CHAR(65+idx) // $CHAR(65)=A
    do arr.%set(idx,val)
}
// Print the array using %get()
for idx=0:1:max {
    set val = arr.%get(idx)
    write "arr("_idx_")="_"_val_"  "
}

```

Prints:

```
arr(0)=A arr(1)=B arr(2)=C arr(3)=D arr(4)=E arr(5)=F
```

2.4 Gateway Reentrancy

Reentrancy means that when InterSystems IRIS code sends a request to Java, the Java code can respond with a request of its own using the same physical connection and context as the original request. The following is a typical call sequence:

- Starting from the InterSystems IRIS side, a connection is made to Java using a %Net.Remote.Gateway object, allowing us to create proxy objects that can make calls to Java methods.
- From a proxy, the Gateway.GatewayContext.getIRIS() method can be called to create an instance of jdbc.IRIS. The IRIS object uses the same connection and context as the proxy, and provides access to all the methods for invoking Native API calls (see “Using the InterSystems Native API for Java”).
- The IRIS object can call back into InterSystems IRIS, where it can get and set global values or call classmethods and functions. Making one of these calls puts us back in InterSystems IRIS code again.
- From here, we can make another inner call back to Java using the same Gateway connection. Method ##class(%Net.Remote.Gateway).%GetContextGateway() returns the Gateway object for the current context.
- Calls back and forth between InterSystems IRIS and Java can go on indefinitely, always using the same connection and context.

Getting a jdbc.IRIS object from GatewayContext()

The following example gets an IRIS object in Java, and uses it to read a global value from the InterSystems database:

```

IRIS native = GatewayContext.getIRIS();
// Read the value of global ^GlobalName("value2")
if ( native != null ) {
    String globalVal = native.getString("GlobalName", "value2");
}

```

If the code is not invoked inside a valid Gateway context, the getIRIS() method will return a null object.

Getting a Gateway context object from within Native API calls

This example gets a %Net.Remote.Gateway object for the current context, and uses it to create a proxy for a Java object in the same context:

```

set gateway = ##class(%Net.Remote.Gateway).%GetContextGateway()
if gateway'=$$$NULLLOREF {
    set objFruit = ##class(%Net.Remote.Object).%New(gateway, "Fruit")
}

```

If the context is not valid, the Gateway object will be \$\$\$NULLLOREF.

Note: In general, reentrancy support is not compatible with prior versions of the Object Gateway .

2.5 The Java Fruit Class

This section lists the Fruit class used by most of the examples in this chapter. This trivial Java class has all the features needed to demonstrate a simple proxy object: static class method **id()**, and property accessors **getFruit()** and **setFruit()**.

The Fruit class

```
package demo;
public class Fruit {
    public static String id() {
        return "This Java class displays your favorite fruit.";
    }
    public String fruit;
    public Fruit () {
        fruit = "Uglyfruit";
    }
    public void setFruit(String newFruit) {
        fruit = newFruit;
    }
    public String getFruit() {
        return "My favorite fruit is "+fruit;
    }
    public static void main(String []args) {
        System.out.println("\nGenerating output from Java class Fruit: ");
        Fruit myFruit = new Fruit ();
        System.out.println(myFruit.getFruit());
    }
}
```


3

Using the Java Gateway in a Production

This chapter describes how to use the Java Gateway in a production. It discusses the following topics:


- [Adding the Java Gateway Business Service](#)
- [Settings for the Java Gateway Business Service](#)
- [Calling Business Service Methods](#)
- [Creating a Business Operation](#)
- [Calling API Methods](#)
- [Using the Command Prompt](#)
- [Using the Java Gateway Wizard](#)
- [Error Checking](#)
- [Troubleshooting](#)

3.1 Adding the Java Gateway Business Service

While it is possible to start the Java Gateway server from the command prompt, the simplest way to use the Java Gateway with a production is to add and configure the `EnsLib.JavaGateway.Service` class as a business service within the production. You can only do this if the Java Gateway server is on the local machine where you are running InterSystems IRIS™.

Otherwise, you need to start the Java Gateway server from the command prompt. For details, see “[Using the Command Prompt](#).”

To add the `EnsLib.JavaGateway.Service` class as a business service in your production, use the **Interoperability > Configure > Production** page of the Management Portal. The following steps summarize the configuration procedure:

1. Click the add icon () next to the **Services** column to start the Business Service Wizard.
2. Click the **All Services** tab, and choose `EnsLib.JavaGateway.Service` as the **Service Class**. You may accept the default values for the other settings.
3. Click **OK** to display the updated production diagram.
4. Click the new Java Gateway business service configuration item and then the **Settings** tab to configure it.

Unlike most business hosts in a production, `EnsLib.JavaGateway.Service` does not handle any production messages.

3.2 Settings for the Java Gateway Business Service

The following settings specific to the Java Gateway service appear on the **Settings** tab. Hover the cursor over any setting name to display its help text as it appears in the *Class Reference* or click the setting name to display the help text in a separate pop-up window.

Address

IP address or name of the machine where the JVM to be used by the Java Gateway Server is located.

Port

Port number to which the Java Gateway connects. The default is 55555.

HeartbeatInterval

Number of seconds between each communication with the Java Gateway to check whether it is active. When enabled, the minimum value is 5 seconds and the maximum value is 3600 seconds (1 hour). The default is 10 seconds. A value of 0 disables this feature.

HeartbeatFailureTimeout

Number of seconds without responding to the heartbeat, to consider that the Java Gateway is in failure state. If this value is smaller than the `HeartbeatInterval` property, the gateway is in failure state every time the Java Gateway communication check fails. The maximum value is 86400 seconds (1 day). The default is 30 seconds.

HeartbeatFailureAction

Action to take if the Java Gateway goes into a failure state. Setting it to Restart (default) causes the Java Gateway to restart. Setting it to Alert generates an alert entry in the Event Log. This is independent of the **Alert on Error** setting.

HeartbeatFailureRetry

Time to wait before retrying the `HeartbeatFailureAction` if the Java Gateway server goes into failure state, and stays in failure state. The default is 300 seconds (5 minutes). A value of 0 disables this feature, meaning that once there is a failure that cannot be immediately recovered, there are no attempts at automatic recovery.

JavaHome

Location of the JVM; use the path you identified in “[Prerequisites](#),” in the previous chapter. (This is the value that you would use for `JAVA_HOME` environment variable). It is used to find the target JVM and assemble the command to start the Gateway.

If there is a default JVM on the machine that is usable without the need to specify its location, you can leave this setting blank.

ClassPath

Class path containing the files to be passed as an argument when starting the JVM. You must include any jar file that define classes you are importing via the Java Gateway. There is no need to include InterSystems' .jar files used by the Java Gateway. If you are specifying file paths containing spaces or multiple files, you should quote the classpath and supply the appropriate separators for your platform.

The following is an example semicolon-separated list of file paths for a Microsoft Windows platform:

```
C:\Library\mygateway.jar; "C:\Jar files\utilities.jar"
```


Note that additional paths for the classpath can be specified in business operations derived from `EnSLib.JavaGateway.AbstractOperation`. See the property `AdditionalPaths` in that class.

JVMArgs

Optional arguments to be passed to the Java Virtual Machine (JVM) to include when assembling the command to start the Java Gateway. For example, you can specify system properties: `Dsystemvar=value` or set the maximum heap size: `Xmx256m` and so on, as needed.

JDKVersion

Version of JDK used to select the intended version of the InterSystems .jar files. It is used to assemble the command to start the Java Gateway. For example: `Java 1.7`

Logfile

Fully qualified name of a file to log all communication between the InterSystems IRIS server and the Java Gateway. Usually this setting should be left blank, except when troubleshooting. These messages include acknowledgment of opening and closing connections to the server, as well as any difficulties encountered in mapping Java classes to InterSystems IRIS proxy classes.

JavaDebug

Allow a Java debugger (such as Eclipse or JSwat) to attach. If True, enables Java debugging. The default is False.

JavaDebugPort

Specify the port on which to listen. The default is 8000.

JavaDebugSuspend

If Yes, suspend the JVM on start to wait for the debugger to attach. The default is No.

Other settings are common to most business services. See “[Settings in All Business Services](#)” in *Configuring Productions*.

Once you have added and configured the Java Gateway business service, it automatically manages the Java Gateway as follows:

- When the production starts, the Java Gateway business service starts an instance of the Java Gateway server, using the settings that you specify on the configuration page.
- When the production receives a signal to stop, the Java Gateway business service attaches to the Java Gateway server and instructs it to stop, as well.

For more information, see `EnSLib.JavaGateway.Service` in the *Class Reference*.

3.3 Calling Business Service Methods

The Java Gateway business service provides methods that you can use to start, connect to, and stop the Java Gateway engine. You can call the following methods from InterSystems IRIS code after you have configured the Java Gateway business service as a member of the production:

- `StartGateway()`
- `ConnectGateway()`
- `StopGateway()`

See the `EnsLib.JavaGateway.Service` entry in the *Class Reference* for details on these methods.

3.3.1 StartGateway() Method

```
EnsLib.JavaGateway.Service:StartGateway(pJavaHome As %String,  
    pClassPath As %String,  
    pJVMArgs As %String,  
    pPort As %String,  
    pLogFile As %String = "",  
    pDebug As %Boolean = 0,  
    pJDKVersion As %String = "",  
    ByRef pDevice As %String = "",  
    pAddress As %String = "127.0.0.1",  
    pCmdLine As %String = "")
```

This class method starts the Java Gateway server using the specified arguments. If *pDebug* is True, then the JVM is started such that a debugger can attach to it. If *pLogFile* specifies a valid file name, then messages regarding gateway activities are written to this file. These messages include acknowledgment of opening and closing connections to the server, and difficulties encountered (if any) in mapping Java classes to InterSystems IRIS proxy classes.

3.3.2 ConnectGateway() Method

```
EnsLib.JavaGateway.Service:ConnectGateway(pEndpoint As %String,  
    ByRef pGateway As EnsLib.JavaGateway.JavaGateway,  
    pDebug As %Boolean = 0,  
    pTimeout As %Integer = 5,  
    pAdditionalPaths As %String = "")
```

This class method connects to the Java Gateway server at the specified *pEndpoint* (hostname:port:namespace) and returns an instance of the `EnsLib.JavaGateway.JavaGateway` class. If *pDebug* is true, then the connection uses a much longer timeout to allow for a Java debugger (such as Eclipse or JSwat) to attach.

3.3.3 StopGateway() Method

```
EnsLib.JavaGateway.Service:StopGateway(pPort As %String,  
    pAddress As %String = "127.0.0.1",  
    pTimeout As %Integer = 5)
```

This class method connects to the Java Gateway server and shuts it down.

3.4 Creating a Business Operation

An abstract business operation is available as a base for building Java Gateway oriented business operations for productions. You can simply subclass the abstract class `EnsLib.JavaGateway.AbstractOperation` and implement the appropriate message handlers.

Call the **GetConnection()** method to verify the connection and always access the Java Gateway connection object via the gateway connection object returned by the **GetConnection()** method. For example:

```
Set tSC = ..GetConnection(.tJavaGateway)  
If $$$ISOK(tSC)  
{  
    // Now, start using the tJavaGateway instance ...  
}
```

This method returns a private gateway connection object to be used with the proxy classes.

You can configure the Java Gateway IP address and port in the business operation settings when you add the business operation to the production. Note that the connection to the Java Gateway instance is made during **OnInit()** and closed in

OnTearDown(). You must override these methods in the business operation class to implement your own setup and tear down procedures.

See the `EnsLib.JavaGateway.AbstractOperation` entry in the *Class Reference* for details on these methods and also the `AdditionalPaths`, `Address`, `ConnectTimeout`, and `Port` properties.

3.5 Calling API Methods

In addition to using `connect`, `disconnect`, and `stop` from the business service, the following methods are also available in the `EnsLib.JavaGateway.JavaGateway` class. You can use them when the business service model is not appropriate for your situation:

The `EnsLib.JavaGateway.JavaGateway` class provides the following types of methods:

- API methods that let you `%Connect()` to the Java Gateway server, `%Disconnect()` from it, and `%Shutdown()` the Java Gateway server.
- The `%Import()` method, which imports Java classes or packages from the JVM and generates all the necessary proxy classes for the InterSystems IRIS side.
- The `%ExpressImport()` method, which combines calls to `%Connect()`, `%Import()`, and `%Disconnect()`.
- The utility methods `%ClassForName()` and `%GetAllClasses()`.

3.5.1 %Connect() Method

```
Method %Connect(host As %String,
               port As %Integer,
               namespace As %String,
               timeout As %Numeric = 5,
               additionalClassPaths As %ListOfDataTypes = "")
  As %Status [ Final ]
```

The `%Connect()` method establishes a connection with the Java Gateway engine. It accepts the following arguments:

Argument	Description
<i>host</i>	Identifies the machine on which the Java Gateway server is running.
<i>port</i>	Port number over which the proxy classes communicate with the Java classes.
<i>namespace</i>	interoperability-enabled namespace.
<i>timeout</i>	Number of seconds to wait before timing out, the default is 5.
<i>additionalClassPaths</i>	Optional — use this argument to supply additional class paths; the paths are added to the system class loader and are available until the session terminates.

3.5.2 %Disconnect() Method

```
Method %Disconnect() As %Status [ Final ]
```

The `%Disconnect()` method closes a connection to the Java Gateway engine.

3.5.3 %Shutdown() Method

```
Method %Shutdown() As %Status [ Final ]
```

The `%Shutdown()` method shuts down the Java Gateway engine.

3.5.4 %Import() Method

```
Method %Import( javaClass As %String,
                ByRef imported As %ListOfDataTypes,
                additionalClassPaths As %ListOfDataTypes = "",
                exclusions As %ListOfDataTypes = "" )
    As %Status [ Final ]
```

The `%Import()` method imports the given *javaClass* and all its dependencies by creating and compiling all the necessary proxy classes. The `%Import()` method returns, by reference, a list (*imported*) of generated InterSystems IRIS proxy classes. For details of how Java class definitions are mapped to InterSystems IRIS proxy classes, see the “[Mapping Specification](#)” chapter.

`%Import()` is a onetime, startup operation. You only need to call it the first time you wish to generate the InterSystems IRIS proxy classes. It is necessary again only if you recompile your Java code and wish to regenerate the proxies.

Note: Though it was necessary in earlier versions, `%Import()` does not need to be called at runtime every time you connect.

The following sections provide more details about the `%Import()` method:

- [Import Arguments](#)
- [Import Dependencies and Exclusions](#)

3.5.4.1 %Import() Arguments

Before you invoke `%Import()`, prepare the `%ListOfDataTypes` arguments *additionalClassPaths* and *exclusions*. That is, for each argument, create a new `%ListOfDataTypes` object and call its `Insert()` method to fill the list.

You can use the optional *additionalClassPaths* argument to supply additional *CLASSPATH* arguments, such as the name of the jar file that contains the classes you are importing via the Java Gateway. List elements should correspond to individual additional class path entries, which require one of the following formats:

```
"rootdirectory\..."
"rootdir\...\myjarfile.jar"
```

The additional paths are added to the system class loader and are available until the session terminates. Wildcards are not accepted in *CLASSPATH* arguments; you must use a full name.

Note: While the examples in this topic use Windows pathname conventions, other supported InterSystems IRIS platforms work also.

3.5.4.2 Import Dependencies and Exclusions

While mapping a Java class into an InterSystems IRIS proxy class and importing it into InterSystems IRIS, the Java Gateway loops over all class dependencies discovered in the given Java class including all classes referenced as properties and in argument lists. In other words, the Java Gateway collects a list of all class dependencies that would be needed for a successful import of the given class, then walks that dependency list and generates all necessary proxy classes.

Important: The Java Gateway import only imports classes, methods, and fields marked as `public`.

You can control this process by specifying a list of package and class name prefixes that you would like to exclude from this process. While this situation is rare, it does give you some flexibility to control what classes get imported. The Java Gateway automatically excludes a small subset of packages such as `sun.*`, `COM.rsa.*`, and most `com.sun.*` packages.

In previous releases, the Java Gateway disallowed import of all `com.sun.*` classes, as some of them are Java internals. However, subsequent releases have relaxed this so that you can import additional `com.sun.*` classes (including `com.sun.tools.javac.Main` and `com.sun.messaging`).

3.5.5 %ExpressImport() Method

```
ClassMethod %ExpressImport(name As %String,
                          port As %Integer,
                          host As %String = "127.0.0.1",
                          silent As %Boolean = 0,
                          additionalClassPaths As %ListOfDataTypes = "",
                          exclusions As %ListOfDataTypes = "")
    As %ListOfDataTypes
```

%ExpressImport() is a one-step convenience class method that combines calls to **%Connect()**, **%Import()**, and **%Disconnect()**. It returns a list of generated proxies. It also logs that list, if you set the *silent* argument to 0. The *name* argument is a semicolon-delimited list of classes or jar files.

3.5.6 %ClassForName() Method

```
Method %ClassForName(className As %String)
    As %Status [ Final ]
```

If you need your ObjectScript code to call the Java method `Class.forName` to load a Java class, use the Java Gateway API method **%ClassForName()** to make the call. Its argument is the name of the class. Use the InterSystems IRIS proxy class name as the argument, rather than the Java class name.

3.5.7 %GetAllClasses() Method

```
Method %GetAllClasses(jarFileOrDirectoryName As %String,
                     ByRef allClasses As %ListOfDataTypes,
                     implements As %String = "")
    As %Status
```

This method returns, in the *ByRef* argument *allClasses*, a list of all public classes available in the jar file or directory specified by the first argument, *jarFileOrDirectoryName*. The *implements* optional argument limits the classes returned to those that are implementations of the class specified by the string. By default, the value is an empty string and the method returns all public classes.

3.6 Using the Command Prompt

Usually you start and stop the Java Gateway server automatically, by configuring the `EnsLib.JavaGateway.Service` business service as a member of the production. Once this is done, the Java Gateway server starts and stops automatically with the production. The **StartGateway()** class method is also available to manually start the Java Gateway server.

However, during development or debugging, or when InterSystems IRIS and the Java Gateway server run on different machines, you may find it useful to start the Java Gateway server from a command prompt. Do this by entering the following command (all on one line). Within this command, the service name has a length limit of 255 characters:

```
java -classpath classpath com.intersystems.gateway.JavaGateway port logfile
```

Argument	Description
<i>classpath</i>	Consists of a semicolon-separated list of paths of the files required to be passed as an argument when starting the JVM, including the jar file that contains the classes you are importing via the Java Gateway. If any path includes space characters, that path should be enclosed within double quotes. Be sure to use pathnames of the appropriate format for your platform.
<i>port</i>	Port number on which to listen for the incoming requests.
<i>logfile</i>	<i>Optional</i> — If specified, the command procedure creates a log file of that name. You must specify the full pathname in the string.

The command line for starting the Java Gateway in JMS mode is similar to the one for starting the Java Gateway, replacing `JavaGateway` with `JMSGateway` in the command (all on one line). Within this command, the service name has a length limit of 255 characters:

```
java -classpath classpath com.intersystems.gateway.JMSGateway port logfile
```

You only need to start the JMS Gateway if you are using the Java Gateway with JMS. Otherwise you can just start the Java Gateway.

3.7 Using the Java Gateway Wizard

You can import a Java class or an entire .jar file using the Java Gateway wizard built into Studio. To start the wizard:

1. Start Studio.
2. From the **Tools** menu, point to and click **Add-Ins**.
3. Click **Java Gateway Wizard** to start the Java Gateway Wizard dialog.
4. Click **Jar File** and either enter the path name or click **Browse** to find the appropriate .jar file.
or
Click **Class Name** and either enter the full path name or click **Browse** to find the appropriate Java class file.
5. Enter the **Host** and **Port** for the Java Gateway server.
6. Enter **Classpaths** and **Exclusions** as instructed in the dialog.
7. If you select a **Jar File** in Step 4, you can click **View** to see a list of the classes in the jar file.
or
If you enter a **Class Name** in Step 4, continue to the next step.
8. Click **Import** to generate InterSystems IRIS proxy classes. The wizard displays the class name as it generates each proxy class.
9. When the import operation is complete, click **Finish** to exit the wizard.

3.8 Error Checking

The Java Gateway provides error checking as follows:

- When an error occurs while executing InterSystems IRIS proxy methods, the error is, in most cases, a Java exception, coming either from the original Java method itself, or from the Java Gateway engine. When this happens, a <ZJGTW> error is trapped.
- Java Gateway API methods like `%Import()` or `%Connect()` return a typical InterSystems IRIS `%Status` variable.

In both cases, InterSystems IRIS records the last error value returned from a Java class (which in many cases is the actual Java exception thrown) in the local variable `%objlasterror`.

You can retrieve the complete text of the error message by calling `$system.OBJ.DisplayError()`, as follows:

```
Do $system.OBJ.DisplayError(%objlasterror)
```

3.9 Troubleshooting

When you encounter problems using the Java Gateway, it is always beneficial to turn on logging. This facilitates InterSystems staff to help you troubleshoot problems. To activate logging, simply identify a log file when you start the Java Gateway. You can do this whether you start from the command line, by configuring the business service, or using the `StartGateway()` business service method.

Sometimes, while using the Java Gateway in a debugging or test situation, you may encounter problems with a Terminal session becoming unusable, or with write errors in the Terminal window. It is possible that a Java Gateway connection terminated without properly disconnecting. In this case, the port used for that connection may be left open.

If you suspect this is the case, to close the port, enter the following command at the Terminal prompt:

```
Close "|TCP|port"
```

Where *port* is the port number to close.

4

Sample Code

You can find sample Java Gateway code in InterSystems IRIS™ installations in the `EnsLib.JavaGateway.Test` class. These samples demonstrate how to generate and use InterSystems IRIS proxy classes. They are presented in the following example sections:

- [Setting Up Java Gateway Examples](#)
- [Running Plain Java Examples](#)
- [Running JDBC Examples](#)
- [Running EJB Gateway Examples](#)
- [Running JMS Gateway Examples](#)
- [Stateless Service Mode Example](#)

For each method described in this chapter, the *port* argument is the port number over which the proxy classes communicate with the Java classes, and *host* identifies the machine on which the Java Gateway server is running. The *port* argument is required; *host* is optional and defaults to "127.0.0.1" (the local machine) if not provided.

4.1 Setting Up Java Gateway Examples

To prepare to run sample code, in each of the examples described in this chapter, you must complete the following steps:

1. Start the Java Gateway server (for JMS, start the JMS Gateway server).
2. Start a Terminal session and change to an interoperability-enabled namespace (for JMS examples you need two Terminal sessions).
3. Make sure to run Import code if this is the first time you are running the sample code or if you have modified or recompiled your Java classes.

To prepare to run any of the sample code located under `EnsLib.JavaGateway.Test` — either for the first time, or after you update or recompile your Java code — you must run the corresponding Import methods found under `EnsLib.JavaGateway.InterfaceEnabler`. This imports the necessary Java classes. The specific sample Import method depends on the type of example you are running. For example:

- To import the sample Java classes provided with InterSystems IRIS:

```
Do ##class(EnsLib.JavaGateway.InterfaceEnabler).ImportJGSamples(port,host)
```

- To import the JDBC interface:

```
Do ##class(EnsLib.JavaGateway.InterfaceEnabler).ImportJDBC(port,host)
```

- To import the InterSystems JBoss Person EJBs, enter the following command (all on one line):

```
Do ##class(EnsLib.JavaGateway.InterfaceEnabler).ImportPersonJBoss(PersonEJBJar,
j2eeJarFile,port,host)
```

Where *PersonEJBJar* points to the *PersonEJB.jar* file (as generated by the InterSystems EJB Boss projection) and *j2eeJarFile* points to the J2EE jar file on your system, for example:

```
c:/myj2ee/j2ee.jar
```

- To import SonicMQ ConnectionFactory (Queue and Topic), allowing connectivity to the SonicMQ JMS Server, enter the following command (all on one line):

```
Do ##class(EnsLib.JavaGateway.InterfaceEnabler).ImportSonicJMS(sonicBrokerJar,
sonicContextJar,port,host)
```

Where *sonicBrokerJar* points to the Sonic *broker.jar* file, for example:

```
C:/Program Files/SonicSoftware/SonicMQ/lib/broker.jar
```

And *sonicContextJar* points to the Sonic *mfcontext.jar* file, for example:

```
C:/Program Files/SonicSoftware/SonicMQ/lib/mfcontext.jar
```

- To import all J2EE interfaces (EJB, JCA, JTA, Java XML, JTA, etc.), enter the following command (all on one line):

```
Do ##class(EnsLib.JavaGateway.InterfaceEnabler).ImportJ2EE(j2eeJarFile,
port,host)
```

Where *j2eeJarFile* points to the J2EE jar file on your system, for example:

```
c:/myj2ee/j2ee.jar
```

In addition to these Import methods, the *EnsLib.JavaGateway.InterfaceEnabler* class provides a convenience method that, given a jar file or a directory name, displays all available classes in that jar file or directory:

```
Do ##class(EnsLib.JavaGateway.InterfaceEnabler).Browse(jarName,port,host)
```

4.2 Running Plain Java Examples

The **Test()** method shows how to use the sample classes delivered with InterSystems IRIS. To run it, first [set up](#) the example, using **ImportJGSamples()** if you need to import. Then enter:

```
Do ##class(EnsLib.JavaGateway.Test).Test(port,host)
```

The **TestArrays()** method shows how to use arrays. To run it, first [set up](#) the example, using **ImportJGSamples()** if you need to import. Then enter:

```
Do ##class(EnsLib.JavaGateway.Test).TestArrays(port,host)
```

4.3 Running JDBC Examples

The following example establishes a connection with InterSystems JDBC driver, then executes some standard JDBC code. To run it, first [set up](#) the example, using **ImportJDBC()** if you need to import. Then enter:

```
Do ##class(EnsLib.JavaGateway.Test).JDBC(port,host,jdbcPort,jdbcHost)
```

This sample code should work against any database that has a compliant JDBC driver. Simply replace the connection parameters (JDBC driver class name, URL, username, and password) with appropriate values. See the *Class Reference* entry for the **JDBC** method for details.

4.4 Running EJB Gateway Examples

The following example shows how ObjectScript code can access a sample Entity Bean (generated by InterSystems IRIS EJB projections) using JBoss version 4.0.1. To run it, first [set up](#) the example, using **ImportPersonJBoss()** if you need to import. Then enter:

```
Do ##class(EnsLib.JavaGateway.Test).PersonJBoss(JBossRoot,port, host)
```

Where *JBossRoot* points to your JBoss root, for example:

```
c:/jboss-4.0.1spl
```

You can easily modify this example to work against *any* application server. Simply set the *CLASSPATH* accordingly and use appropriate connection and context parameters.

4.5 Running JMS Gateway Examples

The following examples show how to use Pub/Sub (Topic) and P2P (Queue) JMS Gateways against the SonicMQ JMS Server. The sample Topic/Queue shipped with SonicMQ is used throughout the examples.

Important: To use the JMS Gateway you must have the `java.jms.*` libraries which are included in the J2EE SDK `j2ee.jar` file.

The examples below can be modified to work against *any* JMS Server. Simply set the *CLASSPATH* accordingly and use appropriate Queue/Topic Connection factories. You must also modify the **ImportSonicJMS()** method to import corresponding connection factories, and you need to use appropriate sample Topics/Queues.

- [JMS Point-to-Point \(P2P\) Example](#)
- [JMS Publish/Subscribe \(Pub/Sub\) Example](#)
- [Java Naming and Directory Interface \(JNDI\) Example](#)

4.5.1 JMS Point-to-Point (P2P) Example

The following example shows how to generate a JMS-based Queue message sender and receiver and control them from an InterSystems IRIS session. To run the example, first [set it up](#), using **ImportSonicJMS()** if you need to import. Then:

1. In one Terminal session, type:

```
Do ##class(EnsLib.JavaGateway.Test).JMSQueueReceiver(sonicBrokerJar,port,host)
```

Where *sonicBrokerJar* points to the Sonic `broker.jar` file, for example:

```
C:/Program Files/SonicSoftware/SonicMQ/lib/broker.jar
```

The example connects to the SonicMQ JMS Server using the sample queue that SonicMQ provides. After that, it waits for incoming messages and displays any received messages on the Terminal screen.

2. In the other Terminal session, type:

```
Do ##class(EnsLib.JavaGateway.Test).JMSQueueSender(sonicBrokerJar,port,host)
```

The example connects to the SonicMQ JMS Server using the sample queue that SonicMQ provides. After that, it sits in a loop, prompting the user to type a message.

3. On the Sender side, you are prompted to type a message.
4. Type a message and press **Enter**.
5. The message appears on the Receiver side.
6. The example runs until you close both Terminal sessions, or until you enter `QUIT` as the message text.

4.5.2 JMS Publish/Subscribe (Pub/Sub) Example

The following example shows how ObjectScript code can generate a JMS-based Topic Publisher and Subscriber and use them to send messages. To run it, first [set up](#) the example, using **ImportSonicJMS()** if you need to import. Then:

1. In one Terminal session, run:

```
Do ##class(EnsLib.JavaGateway.Test).JMSSubscriber(sonicBrokerJar,port,host)
```

Where *sonicBrokerJar* points to the Sonic `broker.jar` file, for example:

```
C:/Program Files/SonicSoftware/SonicMQ/lib/broker.jar
```

The example connects to the SonicMQ JMS Server using the sample topic that SonicMQ provides. After that, it establishes a topic connection and waits to receive a single message.

2. In the other Terminal session, run:

```
Do ##class(EnsLib.JavaGateway.Test).JMSPublisher(sonicBrokerJar,port,host)
```

The example connects to the SonicMQ JMS Server using the sample topic that SonicMQ provides. After that, it establishes a topic connection and creates and sends a single message.

3. The publisher sends one message to the subscriber ("Hello JMS!") and both quit.

4.5.3 Java Naming and Directory Interface (JNDI) Example

This example is exactly the same as the JMS Pub/Sub example, except that the **JNDIPublisher()** method establishes a connection by looking up a *TopicConnectionFactory* object in the JNDI registry, whereas **JMSPublisher()** actually creates the connection. If **JNDIPublisher()** does not find the object, it tries to (re)bind the object, then return and try to look it up.

Note: The JMS with JNDI example is customized for JMS. However, you can use JNDI to obtain a factory object in other contexts, such as with Enterprise Java Beans (EJB).

To run the example, first [set it up](#), using **ImportSonicJMS()** if you need to import. Then:

1. In one Terminal session, type:

```
Do ##class(EnsLib.JavaGateway.Test).JMSSubscriber(sonicBrokerJar,port,host)
```

Where *sonicBrokerJar* points to the Sonic `broker.jar` file, for example:

```
C:/Program Files/SonicSoftware/SonicMQ/lib/broker.jar
```

2. In the other Terminal session, type (all on one line):

```
Do ##class(EnsLib.JavaGateway.Test).JNDIPublisher(sonicBrokerJar,
sonicContextJar,port,host)
```

Where *sonicContextJar* points to the Sonic *mfcontext.jar* file, for example:

```
C:/Program Files/SonicSoftware/SonicMQ/lib/mfcontext.jar
```

3. The publisher sends one message to the subscriber ("Hello JMS!") and both exit.

4.6 Stateless Service Mode Example

Here is a simple implementation using GSON which gets the Google directions between two cities and sends them back in InterSystems IRIS JSON format. For more info on GSON go to: <https://code.google.com/p/google-gson/>.

Java code:

```
package jsonservice;
import java.io.BufferedReader;
import java.io.ByteArrayInputStream;
import java.io.InputStreamReader;
import java.net.URL;

import com.google.gson.JsonElement;
import com.google.gson.JsonObject;
import com.google.gson.JsonParser;

public class Directions implements com.intersystems.gateway.Service {

    public byte[] execute(byte[] args) throws Throwable {
        JsonElement inputJSON = new JsonParser().parse(new
            BufferedReader(new InputStreamReader(new ByteArrayInputStream(args),
"UTF-8"))));
        JsonObject jsonObject = inputJSON.getAsJsonObject();
        String origin = jsonObject.get("origin").toString();
        String destination = jsonObject.get("destination").toString();

        URL URLsource = new URL("http://maps.googleapis.com/maps/api/directions/json?
            origin="+origin+"&destination="+destination+"&sensor=false");
        BufferedReader in = new BufferedReader(new InputStreamReader(URLsource.openStream(),"UTF-8"));

        JsonElement outputJSON = new JsonParser().parse(in);
        in.close();
        jsonObject = outputJSON.getAsJsonObject();
        String response = jsonObject.toString();
        return response.getBytes();
    }
}
```

To invoke the above service from InterSystems IRIS, simply do:

```
Set classPath=##class(%ListOfDataTypes).%New()
// add GSON to the classpath
Do classPath.Insert("c:/service/gson-1.4.jar")
// add the location of the above Service to the classpath
Do classPath.Insert("c:/service/")
// invoke the service
Write ##class(%Net.Remote.Gateway).%RemoteService("127.0.0.1",55555,"jsonservice.Directions",{"origin"
:
    "philadelphia", "destination" :
    "boston"}",classPath)
```

Note: The JSON parsing tool (GSON) is not strictly necessary in this simple example.

5

Mapping Specification

This chapter describes the mapping between Java objects and the InterSystems IRIS™ proxy classes that represent the Java objects.

Important: Only classes, methods, and fields marked as `public` are imported.

This chapter describes mappings of the following types:

- [Package and Class Names](#)
- [Primitives](#)
- [Date and Time](#)
- [Properties](#)
- [Methods](#)
- [Constructors](#)
- [Constants](#)
- [Java Classes](#)
- [Restrictions](#)

5.1 Package and Class Names

Package and class names are preserved when imported, except that each underscore (`_`) in an original Java class name is replaced with the character `u` and each dollar sign (`$`) is replaced with the character `d` in the InterSystems IRIS proxy class name. Both the `u` and the `d` are case-sensitive (lowercase).

5.2 Primitives

Primitive types and primitive wrappers map from Java to InterSystems IRIS as shown in the following table.

Java	InterSystems IRIS
<code>boolean</code>	<code>%Library.Boolean</code>

Java	InterSystems IRIS
byte	%Library.Integer
char	%Library.String
double	%Library.Numeric
float	%Library.Double
int	%Library.Integer
long	%Library.Integer
short	%Library.SmallInt
java.lang.Boolean	%Library.Boolean
java.lang.Double	%Library.Numeric
java.lang.Float	%Library.Double
java.lang.Integer	%Library.Integer
java.lang.Long	%Library.Integer
java.lang.Short	%Library.SmallInt
java.lang.String	%Library.String

Primitive Java type wrappers are mapped by default to their corresponding InterSystems IRIS data types for performance reasons. It is recommended that you always use data types whenever you are passing an argument whose type is a primitive wrapper. For example, you can call the following Java method:

```
public Long getOrderNumber(Integer id, Float rate)
```

as follows in InterSystems IRIS:

```
Set id=5
Set rate=10.0
// order is a local variable
Set order=test.getOrderNumber(id,rate)
```

However, you are also free to import primitive wrapper types as is, then use them that way from your InterSystems IRIS code, for example:

```
Set id=##class(java.lang.Integer).%New(gateway,5)
Set rate=##class(java.lang.Float).%New(gateway,10.0)
// order is of java.lang.Long type
Set order=test.getOrderNumber(id,rate)
```

5.3 Date and Time

Date and time types map from Java to InterSystems IRIS as follows:

Java	InterSystems IRIS
java.sql.Date	%Library.Date
java.sql.Time	%Library.Time
java.sql.Timestamp	%Library.TimeStamp

5.4 Properties

The result of importing a Java class is an ObjectScript abstract class. For each Java property that does not already have corresponding getter and setter methods (imported as is), the Java Gateway engine generates corresponding ObjectScript getter and setter methods. It generates setters as `setXXX`, and getters as `getXXX`, where `XXX` is the property name. For example, importing a Java string property called `Name` results in a getter method `getName()` and a setter method `setName(%Library.String)`. The gateway also generates set and get class methods for all static members.

5.5 Methods

After you perform the Java Gateway import operation, all methods in the resulting InterSystems IRIS proxy class have the same name as their Java counterparts, subject to the limitations described in the [Method Names](#) section. They also have the same number of arguments. The type for all the InterSystems IRIS proxy argument methods is `%Library.ObjectHandle`; the Java Gateway engine resolves types at runtime.

For example, the Java method `test()`:

```
public boolean checkAddress(Person person, Address address)
```

is imported as:

```
Method checkAddress(p0 As %Library.ObjectHandle,
                   p1 As %Library.ObjectHandle) As %Library.ObjectHandle
```

5.5.1 Overloaded Methods

While ObjectScript does not support overloading, you can still map overloaded Java methods to InterSystems IRIS proxy classes. This is supported through a combination of largest method cardinality and default arguments. For example, if you are importing an overloaded Java method whose different versions take two, four, and five arguments, there is only one corresponding method on the InterSystems IRIS side; that method takes five arguments, all of `%ObjectHandle` type. You can then invoke the method on the InterSystems IRIS side with two, four, or five arguments. The Java Gateway engine then tries to dispatch to the right version of the corresponding Java method.

While this scheme works reasonably well, avoid using overloaded methods with the same number of arguments of similar types. For example, the Java Gateway has no problems resolving the following methods:

```
test(int i, String s, float f)
test(Person p)
test(Person p, String s, float f)
test(int i)
```

However, avoid the following:

```
test(int i)
test(float f)
test(boolean b)
test(Object o)
```

Tip: For better results using the Java Gateway, use overloaded Java methods only when absolutely necessary.

5.5.2 Method Names

InterSystems IRIS has a limit of 31 characters for method names. Ensure your Java method names are not longer than 31 characters. If the name length is over the limit, the corresponding InterSystems IRIS proxy method name contains only the first 31 characters of your Java method name. For example, if you have the following methods in Java:

```
thisJavaMethodHasAVeryVeryLongName(int i) // 34 characters long
thisJavaMethodHasAVeryVeryLongNameLength(int i) // 40 characters long
```

InterSystems IRIS imports only one method with the following name:

```
thisJavaMethodHasAVeryVeryLongN // 31 characters long
```

The Java reflection engine imports the first one it encounters. To find out which method is imported, you can check the InterSystems IRIS proxy class code. Better yet, ensure that logging is turned on before the import operation. The Java Gateway log file contains warnings of all method names that were truncated or not imported for any reason.

Each underscore (_) in an original method name is replaced with the character `u` and each dollar sign (\$) is replaced with the character `d`. Both the `u` and the `d` are case-sensitive (lowercase). If these conventions cause an unintended overlap with another method name that already exists on the InterSystems IRIS side, the method is not imported.

Finally, InterSystems IRIS class code is not case-sensitive. So, if two Java method names differ only in case, InterSystems IRIS only imports one of the methods and writes the appropriate warnings in the log file.

5.5.3 Static Methods

Java static methods are projected as class methods in the InterSystems IRIS proxy classes. To invoke them from ObjectScript, use the following syntax:

```
// calls static Java method staticMethodName(par1,par2,...)
Do ##class(className).staticMethodName(gateway,par1,par2,)
```

5.6 Constructors

You invoke Java constructors by calling `%New()`. The signature of `%New()` is exactly the same as the signature of the corresponding Java constructor, with the addition of one argument in position one: an instance of the Java Gateway. The first thing `%New()` does is to associate the proxy instance with the provided gateway instance. It then calls the corresponding Java constructor. For example:

```
// calls Student(int id, String name) Java constructor
Set Student=##class(javagateway.Student).%New(Gateway,29,"John Doe")
```

5.7 Constants

The Java Gateway projects and imports Java static final variables (constants) as Final Parameters. The names are preserved when imported, except that each underscore (_) is replaced with the character `u` and each dollar sign (\$) is replaced with the character `d`. Both the `u` and the `d` are case-sensitive (lowercase).

For example, the following static final variable:

```
public static final int JAVA_CONSTANT = 1;
```

is mapped in ObjectScript as:

```
Parameter JAVAuCONSTANT As INTEGER = 1;
```

From ObjectScript, access the parameter as:

```
##class(MyJavaClass).%GetParameter("JAVAuCONSTANT")
```

5.8 Java Classes

The following sections describe the particulars of using the InterSystems IRIS Java Gateway with specific types of Java classes:

- [Java Object Superclass \(java.lang.Object\)](#)
- [Java Arrays](#)
- [Java Collections Framework](#)
- [Recasting](#)
- [Java Standard Output Redirection](#)

5.8.1 Java Object Superclass (java.lang.Object)

Earlier versions of the Java Gateway did not allow the use of java.lang.Object. This release maps java.lang.Object as is. When using java.lang.Object, consider the following:

- Primitive wrapper classes in Java, which are subclasses of java.lang.Object in Java, are mapped to InterSystems IRIS data types and are thus not subclasses of java.lang.Object in InterSystems IRIS. For details, see the [Java Arrays](#) section.
- Although using java.lang.Object in Java provides great flexibility, it often requires much (re)casting. ObjectScript has only limited support for casting and recasting. When using java.lang.Object to point to its subclass, use the cast operation in ObjectScript to execute the methods of the subclass. Here is an example from the EJB Gateway:

```
Set jndiContext=##class(javax.naming.InitialContext).%New(gateway)
Set jndiName="PersonEJB_Sample_EJBPerson"
Set refPerson=jndiContext.lookup(jndiName)
Set personHomeClass=##class(java.lang.Class).forName(gateway,
Sample.EJBPersonHome)
Set homePerson=##class(javax.rmi.PortableRemoteObject).narrow(gateway,
refPerson,personHomeClass)
// here homePerson is java.lang.Object, and in Java, we would simply
// recast it to EJBPersonHome by saying:
//   homePerson = (EJBPersonHome) homePerson

// In ObjectScript, you will need to 'recast' the method call:
Set remotePerson=##class(Sample.EJBPersonHome)homePerson.findById(1)
```

Using java.lang.Object works as long as you remember you cannot recast an object *per se*. However, since InterSystems IRIS proxy classes are abstract classes, method invocation recasting is sufficient for most purposes.

5.8.2 Java Arrays

Arrays of primitive types, wrappers, data and time types, and Class types are mapped as %Library.ListOfDataTypes. Arrays of object types are mapped as %Library.ListOfObjects. Only one level of subscripts is supported.

Java byte arrays (byte[]) are projected as %Library.GlobalBinaryStream. Similarly, Java char arrays (char[]) are projected as %Library.GlobalCharacterStream. This allows for a more efficient handling of byte and character arrays.

As an only exception to the general rule of pass-by-value-only semantics in the Java Gateway, you can pass byte and stream arrays either by value or by reference. Passing by reference allows changes to the byte/char stream on the Java side visible on the InterSystems IRIS side as well. A good example is the **java.io.InputStream** read method:

```
int read(byte ba[], int maxLen
```

which reads up to *maxLen* bytes into the *ba* byte array. For example, in Java:

```
byte[] ba = new byte[maxLen];
int bytesRead = inputStream.read(ba,maxLen);
```

The equivalent code in ObjectScript:

```
Set readStream=##class(%GlobalBinaryStream).%New()
// reserve a number of bytes since we are passing the stream by reference
For i=1:1:50 Do readStream.Write("0")
Set bytesRead=test.read(.readStream,50)
```

The following example passes a character stream by value, meaning that any changes to the corresponding Java char[] are not reflected on the InterSystems IRIS side:

```
Set charStream=##class(%GlobalCharacterStream).%New()
Do charStream.Write("Global character stream")
Do test.setCharArray(charStream)
```

5.8.3 Java Collections Framework

Previous versions of the Java Gateway provided special treatment when importing `java.util.List` (and its subclasses), `java.util.Map` (and its subclasses) and `java.util.Class`. The Java Gateway imported the first two as either `%Library.ListOfDataTypes` or `%Library.ListOfObjects` and `java.util.Class` as `%Library.String`.

This release now imports all of the above classes “as is.” You now can use the entire Java Collections Framework “as is” in InterSystems IRIS. You can also take advantage of `java.lang.Class` methods. The following is a **HashMap** example using ObjectScript:

```
Set grades=##class(java.util.HashMap).%New(gateway)
Set x=grades.put("Biology",3.8)

Set x=grades.put("Spanish",2.75)
Do student.mySetGrades(grades)

Set grades=student.myGetGrades()
Set it=grades.keySet().iterator()
While (it.hasNext()) {
    Set key=it.next()
    Set value=grades.get(key)
    Write " ",key," ",value,!
}
```

The following example uses `Class.forName` and `java.util.ArrayList`:

```
Set arrayListCls=##class(java.lang.Class).forName(gateway,"java.util.ArrayList")
Set sports=arrayListCls.newInstance()
Do sports.add("Basketball")

Do sports.add("Swimming")

Set list=student.myGetFavoriteSports()
For i=0:1:list.size()-1 {
    Write " "_list.get(i),!
}
```

5.8.4 Recasting

ObjectScript has limited support for recasting; namely, you can recast only at a point of a method invocation. However, since all InterSystems IRIS proxies are abstract classes, this should be quite sufficient. For an example of how to recast, see the [Java Object Superclass](#) section.

5.8.5 Java Standard Output Redirection

The Java Gateway automatically redirects any standard Java output in the corresponding Java code to the calling InterSystems IRIS session. It collects any calls to `System.out` in your Java method calls and sends them to InterSystems IRIS to display in the same format as you would expect to see if you ran your code from Java. To disable this behavior and direct your output to the standard output device as designated by your Java code (in most cases that would be the console), set the following global reference in your interoperability-enabled namespace:

```
Set ^%SYS("Gateway", "Remote", "DisableOutputRedirect") = 1
```

5.9 Restrictions

Important: Rather than aborting import, the Java Gateway engine silently skips over all the members it is unable to generate. If you repeat the import step with logging turned on, InterSystems IRIS records all skipped members (along with the reason why they were skipped) in the `WARNING` section of the log file.

The Java Gateway engine always makes an attempt to preserve package and method names, parameter types, etc. That way, calling an InterSystems IRIS proxy method is almost identical to calling the corresponding method in Java. It is therefore important to keep in mind ObjectScript restrictions and limits while writing your Java code. In a vast majority of cases, there should be no issues at all. You might run into ObjectScript limits if, for example:

- Your Java method names are longer than 30 characters.
- You have 100 or more arguments.
- You are trying to pass String objects longer than 32K.
- You rely on the fact that Java is case-sensitive when you choose your method names.
- You are trying to import a static method that overrides an instance method.

Check with the latest ObjectScript documentation regarding any limits or restrictions. The books are:

- *Using ObjectScript*
- *ObjectScript Reference*

