WebSphere Application Server for z/OS V6.1

WSADMIN Primer

(With ***Jython*** *Scripting Illustrated)*

This document supersedes WP100421, which was V5.x and JACL

This document can be found on the web at:
`www.ibm.com/support/techdocs`
Search for document number **`WP101014`** under the category of "White Papers"

Version Date: September 10, 2008
Please see "Document Change History" on page 108 for updates provided in this version of the document.

IBM Washington Systems Center

Don Bagwell
IBM Washington Systems Center
301-240-3016
`dbagwell@us.ibm.com`

Many thanks to **Mike Loos** for sharing with me his knowledge of WSADMIN, in particular the specifics of Jython and the "fine grain security" topic. **Mike Kearney** is my source for nearly all things related to security, such as keyrings, certificates and all that other mysterious stuff. **Rohith Ashok** of WebSphere development, who seems to know something about any question I've ever asked him.

Table of Contents

Introduction and Overview

In this section we'll explore some high-level topics and set the stage for the more detailed exploration of WSADMIN that follows.

What is WSADMIN?

You can think of WSADMIN as a programming interface to the configuration and management of a WebSphere Application Server cell. With WSADMIN you can programmatically do pretty much anything you can do in the Admin Console.

What is the primary benefit of WSADMIN?

One of the most common uses of WSADMIN is to automate the installation of applications into WebSphere. The benefit is that it provides a way to make certain an application is installed in a consistent way in different environments, such as development, QA, test and then production.

That basic notion -- what would normally be done in the Admin Console is now done with a program and WSADMIN -- provides a way to automate and streamline WebSphere configuration and operations.

Is this just for WebSphere on z/OS?

No. WSADMIN is a common feature of WebSphere Application Server across all platforms. There are some aspects of WSADMIN usage that are unique to z/OS, but the vast majority of WSADMIN is common across the platforms.

That said, in this document we'll highlight the z/OS-specific things, particularly as it relates to things like how to invoke WSADMIN using batch JCL, and security issues related to RACF.

Basic elements of WSADMIN

Here we'll introduce you to three of the basic pieces of the WSADMIN puzzle. You'll get to see how these things work as you go through the exercises provided.

Five programming "objects" (or "commands")

With WebSphere Application Server V6.1, WSADMIN now comes with five programming "objects".

??? Don't worry if you're not familiar with object oriented programming. Think of an "object" in this sense as a "command."

Object	*Description*
`AdminApp`	Used for application-oriented things, such as installing or uninstalling an application.
`AdminConfig`	Used for configuration-oriented things, such as adding a new server, or changing the name of something in the configuration.
`AdminControl`	Used to control the running WebSphere environment, such as starting and stopping servers, or synchronizing changes to a node.
`AdminTask`	New with V6.1, this object provides a simplified way to do what used to be much more complicated. For example, changing the IP host name and MVS system name throughout a node. In the past that was a very challenging thing to do, but with AdminTask it becomes a single command.
`Help`	As the name implies, this is a help facility that is quite handy in determining the syntax requirements of the other objects.

Note: We've intentionally glossed how each object has "methods" (sub-commands), parameters and options. Learning the syntax of these WSADMIN objects can be the most challenging aspect of learning WSADMIN. We'll get to all that. These "objects" are really the major high-level commands.

WSADMIN "client"

In order to get your commands processed against the interface of WebSphere, you'll need to invoke the WSADMIN "client". You provide the client the WSADMIN commands; the client then works against the programming interface of WebSphere.

The WSADMIN client is really just a UNIX shell script. It's a *very smart* shell script, but still just a shell script you invoke. That shell script is found in the configuration directory of your WebSphere for z/OS server. The name of the shell script is `wsadmin.sh`, and is found under:

```
/<mount_point>/DeploymentManager/profiles/default/bin
```

The way that shell script is invoked, and how WSADMIN commands are processed by it is what this white paper is all about.

Programming "script"

WSADMIN is more properly referred to as a "scripting interface" because it can be used for more than issuing *just* those five objects we listed before. We may also use a programming language with it. Using a programming language gives us more power to do things like use variables, if-then-else structures, and loops.

For WSADMIN we have two programming languages we may use -- Jython and JACL. Both are more properly called *interpreted scripts*. That's why WSADMIN is called a "scripting interface." We don't need to compile Jython or JACL. We simply tell the WSADMIN client to *read and interpret* the script we provide it.

This document will focus on *Jython* because it's the preferred scripting language. JACL was what WSADMIN supported initially. But it is now deprecated in V6.1. That means it still works, but eventually will be dropped from support. New development should be done in Jython.

Notes:
- No, unfortunately REXX is not a supported scripting language.
- People with a library of JACL scripts might wish to consider the "IBM Jacl to Jython Conversion Assistant," a tool to assist in the conversion of Jacl to Jython. That can be found at:

  `http://www.ibm.com/support/docview.wss?rs=180&uid=swg24012144`

Here's a very simple a Jython script used to install an application:

```
  ear     = '/u/myapps/SuperSnoopProj.ear'
1 node    = 'mynodec'
  server  = 'mysr01c'               2
  options = '[-node ' + node + ' -server ' + server + ']'
  AdminApp.install(ear,options)
3 AdminConfig.save()
```

It's too early to give a full description of all that's going on in that example, but for the sake of getting a sense of it, here are some notes:

1. This script creates four variables right up front -- `ear`, `node`, `server`, `options`. The use of variables is simply a way to code custom information in one place in the script, then use those variables in lots of different places throughout the script.
2. The variable options is a special type of variable. Think of it as like an array ... a way to hold information in a variable so that each element can be referenced directly, without having to do things like sub stringing some number of bytes in. The WSADMIN objects require that any option lists be in an array-like format, so to accomplish that we use this special kind of variable construction. We'll see more of that later.
3. Two WSADMIN objects are used -- `AdminApp` is what installs the application into WebSphere; `AdminConfig` is what's used to save the changes. You'll notice that each has something

appended to it -- `AdminApp.install` and `AdminConfig.save`. Those are *methods* on the objects. `AdminApp` has several different methods, `install` being one of them. Think of methods as sub-commands.

Scripts can be a great deal more sophisticated than that. But this simple example illustrates a key point -- the Jython script language is the program that works against the programming interface (that's what WSADMIN really is) of WebSphere.

How the WSADMIN client is invoked

As mentioned, the client is a UNIX shell script. That means it can be invoked in two basic ways:

- *Command Line* -- you, the operator, invokes the `wsadmin.sh` shell script by hand. This command line may be a Telnet session, or an OMVS session.
- *Batch JCL* -- using `BPXBATCH`, which invokes the `wsadmin.sh` shell script from within JCL.

The two end up doing essentially the same thing. However, there are some differences. For instance, the command line invocation allows you to issue WSADMIN commands interactively, while the batch JCL method is just that ... batch. Both have their places.

We have much more to cover on the subject of invoking the `wsadmin.sh` shell script client.

Important Point: two "modes" -- local and remote

The WSADMIN client has two basic "modes" in which it can operate: *remote* and *local*.

- *Remote mode* -- the `wsadmin.sh` client establishes a *TCP network connection* with the running Deployment Manager (or Standalone server).
- *Local mode* -- no TCP connection is made; the `wsadmin.sh` client *directly* manipulates the configuration XML files.

There's a lot of detail yet to be discussed. For now, let's summarize some of the differences between the two:

	Pros	*Cons*
Remote Mode	• WSADMIN client may be anywhere: the same LPAR as WebSphere, a different LPAR, a different Sysplex, or even on a distributed platform like Windows or UNIX. • If someone else is using the Admin Console at the same time, WebSphere is able to juggle the two so changes aren't in conflict. • The "fine grained security" capabilities of WebSphere V6.1 may be used [1]	• WebSphere must be up and running for this to work • The security implications of SSL across the network connection can be confusing
Local Mode	• The issue of coordinating SSL certificates isn't present because no network connection is made. • WebSphere does not need to be active for this to work.	• The only security that comes into play is file access permissions. • Using this when someone else is using the Admin Console can create problems. WSADMIN is directly manipulating XML files and the Admin Console will report (or warn) that its configuration base is changing unexpectedly.

Basic Rule: If the Deployment Manager (or Standalone server) is running, then use "remote mode" and connect to the DMGR so it can manage the updates. Use "local mode" only if the server is down.

[1] What the "fine grained security" provides is a way to limit what parts of a WebSphere configuration a given user is permitted to act upon. For instance, "Fred" is allowed to install applications into server XYZ only, but no others. More on fine grained security and WSADMIN later.

This document is not the complete source of information on WSADMIN

The topic is simply too big. Rather than try to be the source of all information, this document is designed to introduce the reader to some key concepts. The thinking is that once you have those key concepts figured out, and you've done a few basic things with WSADMIN, you'll be in a better position to use the other sources of information out there.

How this document is organized

This white paper is intended to be worked through from start to finish, with you performing the simple exercises along the way. The exercises are designed to introduce key WSADMIN concepts to you in a logical, systematic way. Intermixed with the exercises are further explanations of important concepts.

Any time you see a square checkbox serving as a bullet to the left of a paragraph, consider it a signal that there's a "to do" for you in that paragraph.

Performing the exercises provided in this document

You will need access to a WebSphere Application Server for z/OS configuration, preferably a "Network Deployment" cell (with a Deployment Manager). This cell should *not* be a production cell as some of the exercises will result in modifications being made to the configuration.

A "standalone server" will also work, unless the nature of the exercise is applicable only to a Network Deployment configuration. Examples of that would be node synchronization and the adding of a new server -- neither of those things applies to a standalone server.

Note: We will generally illustrate the use of a Deployment Manager. You'll see that we'll reference the `/DeploymentManager` directory. If you have a standalone server, substitute `/AppServer` for `/DeploymentManager`. Everything else will be the same.

Other sources of WSADMIN information

- `ibm.com/support/techdocs/atsmastr.nsf/WebIndex/WP100421`

  This is the older V5 edition of this Primer, written to the JACL scripting standard. The document you're reading right now is intended to replace WP100421.
- `ibm.com/support/techdocs/atsmastr.nsf/WebIndex/WP100963`

  This is the Jython white paper written by Mike Loos. It has some more sophisticated scripting examples than does this document. Don't overlook that as a good source of information just because this new Primer has been published.
- The InfoCenter, which is the best place to go for reference information on command syntax and usage.

Summary

WSADMIN is a programming interface to WebSphere Application Server's management and configuration function. You use a program -- a "script" -- to issue commands against the API and thus make the changes you wish to make. WSADMIN can be used to do most everything that can be done through the Admin Console. The primary benefit of WSADMIN is that tasks can be automated, making repetitive tasks more easily done and done consistently.

You may be tempted to get right into the scripting that modifies the configuration. But we believe there is great value in understanding the other basics of this thing, such as how to invoke WSADMIN, and how to program with Jython. Be patient with your learning, and follow the exercises from front to back. It'll be worth it.

WSADMIN Invocation Exercises

Lesson Overview

In this section we'll perform exercises related to the invoking of WSADMIN from a Telnet (or OMVS) session. Contrast this with invoking WSADMIN from batch JCL, which we cover under "JCL and BPXBATCH Exercises" starting on page 34. The two are very similar to one another, but there are aspects of batch invocation we wanted to highlight so we broke it out into a separate section.

We won't do much with WSADMIN in this section. We'll just explore the different ways to invoke the client. This will lay the groundwork for the actual Jython scripting elsewhere in this document.

Checklist: what you need to do the exercises in this section

Make sure you have the following available to you:

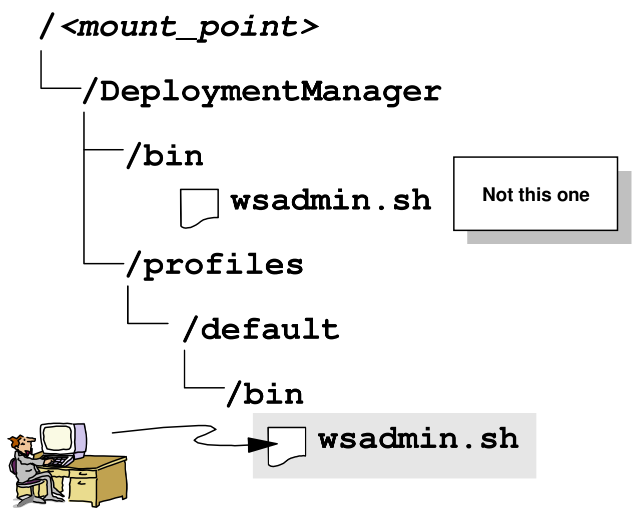
- ○ A WebSphere for z/OS Network Deployment configuration, with Deployment Manager running, and one in which it's okay for you to be making the small changes that may occur from these exercises.

  Note: A standalone server configuration will suffice for these exercises.

- ○ A Telnet or OMVS session. (Telnet is preferable because pasting a long command into OMVS can be cumbersome, while most Telnet clients will wrap the long line for you.)
- ○ Access to the WebSphere "Admin ID" for that cell, along with that ID's password.
- ○ Access to the Admin Console
- ○ Knowledge of the what the DMGR (or standalone server) configuration mount point is.

Background: where `wsadmin.sh` client shell script is located

There are actually two copies in each node, but the `wsadmin.sh` client shell script that you are to invoke is located in the following place:



Note: Or under `/AppServer` if a standalone configuration.

The one located up under `/bin` might also work, but the rule is always invoke the one under `/profiles/default/bin`. The one under `/profiles/default/bin` establishes the "command line environment" (including `STEPLIB` if applicable) then it calls the one under `/bin`.

Again, the rule is: always invoke the one under `/profiles/default/bin`.

Exercise: invoke WSADMIN with no parameters at all

Objective	The purpose of this exercise is to show you what happens when `wsadmin.sh` is invoked without any parameters. Then we'll start exploring some of the parameters that determine how client operates.

- ☐ Open up a Telnet session (or OMVS session ... but *not* ISHELL)
- ☐ Log on with the WebSphere Admin ID. Or log on with some other ID and then "switch user" to the WebSphere Admin ID.

Why? That ID has the necessary file system read and write permissions to invoke the WSADMIN client. Some other ID might not, which would result in a failure. It is **not** recommended you do this with a UID=0 ID. That might result in some changed file ownerships, which could cause problems later. Better to operate under the WebSphere Admin ID.

The other reason you want to use the WebSphere Admin ID when first learning WSADMIN is that when security is enabled for a cell (the typical "default" for a V6.1 cell), the WebSphere Admin ID will have the necessary keyring and certificates to establish the SSL connection a connection is made to the running server.

- ☐ Issue the command **`whoami`**. This should reply with your WebSphere Admin ID.
- ☐ Change directories to your Deployment Manager's `/profiles/default/bin` directory

Hint: If you find your system does not display the current directory and you've lost track of where you are, issue the command `pwd`. That stands for "present working directory" and it will report where you are in the configuration structure.

- ☐ Issue the command **`ls`** -- that will result in all the files in that directory being listed out. There's a bunch. You should see `wsadmin.sh` in that list.
- ☐ Now invoke the shell script with no parameters:

```
./wsadmin.sh
```

Note: This might take a minute or so to initialize. Be patient.

What happens next depends on whether security is enabled or not for the cell:

Security Enabled

You'll see something like this:

```
Realm/Cell Name: <default>
Username:
```

When the Admin ID is entered you then see:

```
Password:
```

When the password is supplied you then see:

```
WASX7209I: Connected to process "dmgr" on node <node> using SOAP
connector;  The type of process is: DeploymentManager

WASX7029I: For help, enter: "$Help help"

wsadmin>
```

Security Not Enabled

There's no need to authenticate, so you simply see this:

```
WASX7209I: Connected to process "dmgr" on node <node> using SOAP
connector;  The type of process is: DeploymentManager
WASX7029I: For help, enter: "$Help help"
wsadmin>
```

☐ Issue the command **`quit`** at the `wsadmin>` prompt. This will exit WSADMIN and return you to the standard UNIX prompt.

Review The default behavior of the `wsadmin.sh` client shell script is to use `-conntype SOAP` when no other parameters are offered. If security is enabled for the cell, then the client shell script prompts for the WebSphere Admin ID's userid and password. With that the client is permitted to authenticate into the SOAP port of the Deployment Manager.

WSADMIN client knew what host and port to use by reading the contents of a key XML file: `wsadmin.properties`. The host, port and protocol to use is specified in there.

In a little bit we'll see how to do the same thing, but do it by providing the explicit `-host` and `-port` parameters. If you ever invoke the WSADMIN client somewhere *other than* the DMGR you wish to connect to, you'll need to tell it where to connect. That's what `-host` and `-port` will do.

Exercise: invoke WSADMIN with `-conntype NONE`

Objective In this exercise you'll invoke the WSADMIN client, but with the explicit `-conntype NONE` parameter. This tells the client to go into "local mode" and directly manipulate the XML files.

☐ You should still be the `/profiles/default/bin` directory of your Deployment Manager (or AppServer if standalone).

☐ Issue the command:

```
./wsadmin.sh -conntype NONE
```

There should be *no* prompting for userids or passwords, whether security is enabled or not. You should see:

```
WASX7357I: By request, this scripting client is not connected to any
server process. Certain configuration and application operations will be
available in local mode.
WASX7029I: For help, enter: "$Help help"
wsadmin>
```

☐ Issue the command **`quit`** to exit the WSADMIN prompt.

Review This told the client shell script to go into "local" mode. That meant any changes to the configuration would be done *directly* by the client shell script. When you invoke a copy of the `wsadmin.sh` shell script in "local mode," all it can do is manipulate the configuration files in that *same configuration directory structure*. But not elsewhere.

There is no security checking beyond that done by UNIX file permissions. Since the configuration files are manipulate directly, and no server process is connected to, there is no authentication and no need to pass userid and password. Even if security is enabled that is true.

Exercise: invoke WSADMIN with `-help` parameter

Objective This will provide a printout of all the input parameters supported by the wsadmin.sh shell script client.

☐ Issue the command:

```
./wsadmin.sh -help
```

There should see the following:

Note: It is **not** important for you to understand all this right now. Just be aware of it.

```
WASX7001I: wsadmin is the executable for WebSphere scripting.
Syntax:

wsadmin
        [ -h(elp)  ]
        [ -?  ]
        [ -c <command> ]
        [ -p <properties_file_name>]
        [ -profile <profile_script_name>]
        [ -f <script_file_name>]
        [ -javaoption java_option]
        [ -lang  language]
        [ -wsadmin_classpath  classpath]
        [ -profileName profile]
        [ -conntype
                SOAP
                        [-host host_name]
                        [-port port_number]
                        [-user userid]
                        [-password password] |
                RMI
                        [-host host_name]
                        [-port port_number]
                        [-user userid]
                        [-password password] |
                NONE
        ]
        [ -jobid <jobid_string>]
        [ -tracefile <trace_file>]
        [ -appendtrace <true/false>]
        [ script parameters ]

Where   "command" is a command to be passed to the script processor;
        "properties_file_name" is a java properties file to be used;
        "profile_script_name" is a script file to be executed before the
                main command or file;
        "script_file_name" is a command to be passed to the script processor;
        "java_option" is a java standard or non-standard option to be passed
                to the java program;
        "language" is the language to be used to interpret scripts;
                supported values are "jacl" and "jython".
        "classpath" is a classpath to be appended to built-in one;
        "-conntype"  specifies the type of connection to be used;
                the default argument is "SOAP"
                a conntype of "NONE" means that no server connection is made
                and certain operations will be performed in local mode;
        "host_name"  is the host used for the SOAP or RMI connection;
                the default is the local host;
        "port_number"  is the port used for the SOAP or RMI connection;
        "userid"  is the userid required when the server is running in
                secure mode;
        "password"  is the password required when the server is running in
                secure mode;
        "script parameters"  is anything else on the command line.  These
                are passed to the script in the argv variable; the number of
                parameters is available in the argc variable.
        "jobid_string" is a jobID string to be used to audit each invocation
                of wsadmin;
        "trace_file" is the log file name and location where wsadmin trace
                output is directed;

If no command or script is specified, an interpreter shell is
created for interactive use. To leave an interactive scripting session,
use the "quit" or "exit" commands.

Several commands, properties files, and profiles may be specified
on a single command line.  They are processed and executed in
order of their specification.
```

Review You'll see some of these used throughout this primer, and that's how you'll learn what some of them do. The lesson here was to know about the `-help` parameter and to know that `wsadmin.sh` has a set of acceptable parameters.

Exercise: invoke WSADMIN with `–conntype SOAP`

Objective	What we'll do here will result in the same thing that happened when we invoked `wsadmin.sh` with no parameters -- that is, it'll go into "remote mode" and connect to the DMGR via SOAP. But rather than allowing the default behavior to apply, we're going to explicitly tell WSADMIN to use SOAP. And we'll tell it the host and port *and* pass in the userid and password.

☐ Log into your Admin Console and then drill down to determine the SOAP port of your Deployment Manager:

- ○ "System Administration" ⇨ "DeploymentManager"
- ○ Look on the right side of screen, under "Additional Properties." You should see a link for "Ports." Click on that link and look for `SOAP_CONNECTOR_ADDRESS`. Note the following:

Host	
Port	

☐ Invoke the `wsadmin.sh` client shell script with the following command:

```
./wsadmin.sh –conntype SOAP –host aaaa –port bbbb –user cccc –password dddd
```

where:

- *`aaaa`* is the host address where the DMGR can be reached
- *`bbbb`* is the SOAP port of the DMGR
- *`cccc`* is the WebSphere Admin ID
- *`dddd`* is the password for the WebSphere Admin ID

Hint: Start a Notepad (or other text editor) session with these commands. It'll save a lot of extra typing.

You should see the following:

```
WASX7209I: Connected to process "dmgr" on node <node> using SOAP
connector;  The type of process is: DeploymentManager

WASX7029I: For help, enter: "$Help help"

wsadmin>
```

Even if security is enabled, you do not see the prompt as we saw before. That's because we passed in the userid and password on the invocation command.

☐ Stay at the `wsadmin>` prompt. Proceed to the next exercise.

Review	When invoking the WSADMIN client in "remote" mode, it's always best to use the explicit `–conntype SOAP` with `–host` and `–port`. Even if on the same LPAR, where coding that isn't strictly required. This provides a nice clear way to saying exactly how WSADMIN is to behave. Invoking with no parameters is okay provided you know what's in the `wsadmin.properties` file, which is where it would get its information when no parameters are present.

Exercise: invoke Help object

Objective	The first WSADMIN object we're going to invoke is the `Help` object. This will simply reply back with a list of ways in which `Help` can be used. It's simple to do and will not result in any changes to the configuration.

☐ Issue the following command at the `wsadmin>` prompt:

```
$Help help
```

Note: That dollar sign out front means the scripting language is JACL. That's the default. We'll see how to change that to Jython in a bit.

You should see something like this:

```
WASX7028I: The Help object has two purposes:

        First, provide general help information for the the objects
        supplied by wsadmin for scripting: Help, AdminApp, AdminConfig,
        and AdminControl.

        Second, provide a means to obtain interface information about
        MBeans running in the system.  For this purpose, a variety of
        commands are available to get information about the operations,
        attributes, and other interface information about particular
        MBeans.

        The following commands are supported by Help; more detailed
        information about each of these commands is available by using the
        "help" command of Help and supplying the name of the command
        as an argument.

attributes              given an MBean, returns help for attributes
operations              given an MBean, returns help for operations
constructors            given an MBean, returns help for constructors
description             given an MBean, returns help for description
notifications           given an MBean, returns help for notifications
classname               given an MBean, returns help for classname
all                     given an MBean, returns help for all the above
help                    returns this help text
AdminControl            returns general help text for the AdminControl object
AdminConfig             returns general help text for the AdminConfig object
AdminApp                returns general help text for the AdminApp object
AdminTask               returns general help text for the AdminTask object
wsadmin                 returns general help text for the wsadmin script
                        launcher
message                 given a message id, returns explanation and
                        user action message
```

☐ Now issue the following command at the `wsadmin>` prompt:

`$Help AdminApp`

Note: Case matters. Type carefully.

That will respond with the "methods" of the `AdminApp` object. You should see something like this:

```
deleteUserAndGroupEntries
                Deletes all the user/group information for all
                the roles and all the username/password information for RunAs
                roles for a given application.
edit            Edit the properties of an application
editInteractive Edit the properties of an application interactively
export          Export application to a file
exportDDL       Export DDL from application to a directory
getDeployStatus Returns the combined Deployment status of the application
help            Show help information

install         Installs an application, given a file name and an option string.

installInteractive
                Installs an application in interactive mode, given a
                file name and an option string.
isAppReady      Checks whether the application is ready to be run
list            List all installed applications
listModules     List the modules in a specified application
options         Shows the options available, for a given file, application,
                or in general.
publishWSDL     Publish WSDL files for a given application
```

```
searchJNDIReferences
                List application that refer to the given JNDIName on a given
node
taskInfo        Shows detailed information pertaining to a given install task
                for a given file
uninstall       Uninstalls an application, given an application name and
                an option string
update          Updates an installed application
updateAccessIDs Updates the user/group binding information with accessID
                from user registry for a given application
updateInteractive       Updates an installed application interactively
view            View an application or module,
                given an application or module name
```

Note: At this time it is **not** important to know what all those things do. Just be aware that the Help object is useful in displaying the available methods on an object.

☐ Issue the command `quit` to exit the WSADMIN prompt.

Review What you did there was invoke the `Help` object twice -- once with the method `help`, which displayed back the various methods on the `Help` object; and once with the `AdminApp` method, which displayed all the methods on the `AdminApp` object.

This seems like a trivial thing to do, but it illustrates a very important thing: the interactive help facility of WSADMIN is a very useful tool to see what methods are available on the various WSADMIN objects. It also illustrates how using WSADMIN interactively can be of use.

Background: the default script language

Do you recall the $ sign on the front of the `Help` object? That means the script language used to invoke the `Help` object was JACL. It turns out JACL is the default scripting language in V6.1 for WSADMIN, despite it being "deprecated". JACL still works.

Note: That's good news for backwards compatibility. Scripts written in the past in JACL keep working. For now. At some point the JACL support may drop away. Hence our encouraging Jython.

Jython *can* be used -- *and we recommend that it becomes your preferred scripting language* -- but it requires you to do something so WSADMIN doesn't expect to see the default JACL language. There are two ways to accomplish this:

1. Provide the `-lang Jython` parameter on the WSADMIN invocation command
2. Change the default language expected by WSADMIN

We'll take a look at both next.

How to tell WSADMIN to use Jython rather than JACL

If you want WSADMIN to expect Jython for a particular invoked session of WSADMIN, but you don't want to change the default language, you can pass the `-lang Jython` parameter into the `wsadmin.sh` shell script:

```
./wsadmin.sh -lang jython -conntype SOAP -host aaaa -port bbbb
                                         -user cccc -password dddd
```

Notes:
- Do **not** issue that command right now. We'll do that in a few moments.
- The command used to invoke WSADMIN may end up being quite long ... longer than can be shown on one line in this document. We may break the line as we did here. But the command is *always* entered as one line, not two.

This does *not* change the default scripting language. This simply changes the language for this session of WSADMIN.

Where the default scripting language is defined to WSADMIN

When you invoke the `wsadmin.sh` shell script from a node's `/profiles/default/bin` directory, WSADMIN goes out to the `/profiles/default/properties` directory and reads the file `wsadmin.properties`. That file contains many values that tell WSADMIN how to behave.

The default scripting language is defined a few lines down in the file:

```
#-------------------------------------------------------------------------
# The defaultLang property determines what scripting language to use.
# Supported values are jacl and jython.
# The default value is jacl.
#-------------------------------------------------------------------------
com.ibm.ws.scripting.defaultLang=jacl
```

If you wanted the default language for WSADMIN invoked from this node to be Jython, you would need to change the property to:

```
com.ibm.ws.scripting.defaultLang=jython
```

Note: Do **not** do that at this point in time. We'll do it later, as part of an exercise.

There are many other properties defined in this file. It's worth browsing it and seeing what's there.

Exercise: invoke WSADMIN so Jython is the expected scripting language

Objective	The first WSADMIN object we're going to invoke is the `Help` object. This will simply reply back with a list of ways in which `Help` can be used. It's simple to do and will not result in any changes to the configuration.

☐ Invoke the `wsadmin.sh` client shell script with the following command:

```
./wsadmin.sh –lang jython –conntype SOAP –host aaaa –port bbbb
                                          –user cccc –password dddd
```

Hint: Again, compose this in Notepad first. Then copy/paste into WSADMIN. It'll save typing.

where:

- *aaaa* is the host address where the DMGR can be reached
- *bbbb* is the SOAP port of the DMGR
- *cccc* is the WebSphere Admin ID
- *dddd* is the password for the WebSphere Admin ID

The *first* time you do this you're going to see a list of messages indicating that various JAR files are being processed. This may take a minute or two:

```
  :
*sys-package-mgr*: processing new jar, '/shared/zWebSphere/V6R1B/lib/startup.jar'
*sys-package-mgr*: processing new jar, '/shared/zWebSphere/V6R1B/lib/bootstrap.jar'
*sys-package-mgr*: processing new jar, '/shared/zWebSphere/V6R1B/lib/j2ee.jar'
  :
```

Note: Subsequent invocations of WSADMIN with `–lang Jython` should not result in this happening. Generally this happens only when Jython is invoked the first time for a node. But it may if the location where these JAR files is cached is cleared.

Ultimately you will see:

```
WASX7031I: For help, enter: "print Help.help()"
wsadmin>
```

Note: See the new format of the `Help` object? That's the Jython format.

☐ Now issue the command:

`print Help.help()`

You should get back exactly what you saw with the JACL `$Help help` command.

☐ Now issue the command:

`print Help.AdminApp()`

Again, you should see what you saw with the JACL `$Help AdminApp` command

☐ Issue the same command again, but this time making `AdminApp` *all lowercase*. This will result in an error, illustrating that WSADMIN is case sensitive:

`print Help.adminapp()`

You should see something like this:

```
WASX7015E: Exception running command: "Help.adminapp()"; exception information:
 com.ibm.bsf.BSFException: exception from Jython:
Traceback (innermost last):
  File "<input>", line 1, in ?
AttributeError: adminapp
```

☐ Issue the command `quit` to exit the WSADMIN prompt.

Review	That exercise was to show that a different scripting language -- Jython -- can be invoked by passing in the `-lang jython` parameter on the `wsadmin.sh` invocation. When that takes place, the syntax of the commands changes slightly. Gone is the dollar sign. Methods are specified as "dot extensions" to the object rather than as a parameter that follows after a blank space. *But the output is the same.* That's the key. `$Help help` and `print Help.help()` produce the same results.

Exercise: invoke WSADMIN using RMI rather than SOAP

Objective	To illustrate how the RMI (Remote Method Invocation) protocol can be used rather than SOAP.

☐ Go back to your Admin Console and capture the "ORB" port of your Deployment Manager:

- ○ "System Administration" ⇨ "DeploymentManager"
- ○ Look on the right side of screen, under "Additional Properties." You should see a link for "Ports." Click on that link and look for `ORB_LISTENER_ADDRESS`. Note the following:

Host	
Port	

☐ Invoke the `wsadmin.sh` client shell script with the following command:

```
./wsadmin.sh -lang jython -conntype RMI -host aaaa -port bbbb
                                            -user cccc -password dddd
```

Hint: Again, compose this in Notepad first. Then copy/paste into WSADMIN. It'll save typing.

where:

- *`aaaa`* is the host address where the DMGR can be reached
- *`bbbb`* is the ORB port of the DMGR
- *`cccc`* is the WebSphere Admin ID
- *`dddd`* is the password for the WebSphere Admin ID

☐ Notice that once the `wsadmin>` prompt appears, it looks pretty much just like it did with SOAP. That's as it should be. The difference between RMI and SOAP is at the lower network protocol layer, not up at the WSADMIN behavior layer.

☐ Issue the command `quit` to exit the WSADMIN prompt.

Review	RMI is a more efficient protocol, particularly for SSL, and firewall configurations are more likely to be already configured to allow RMI-IIOP than SOAP. For the purposes of this document we'll just assume you use whichever you prefer. At the higher WSADMIN programming level it doesn't matter.

Lesson wrap-up and summary

In this lesson we went through a few of the essential points about invoking the WSADMIN client. We didn't really do much in the way of actual WSADMIN scripting. Nothing we did changed the configuration at all.

Being able to get to the `wsadmin>` command prompt is a fundamental first step in WSADMIN operations. Once there, then we can start invoking specific WSADMIN objects, make use of various Jython things, and even fetch in Jython scripts kept in files. Invoking WSADMIN from a JCL batch file is really pretty much the same thing we did here, except it's done using BPXBATCH from JCL.

We've intentionally glossed over a few things. This was so this first set of exercises didn't get too complicated. Some of the things we glossed over were:

- The use of RMI vs. SOAP. Those are communication protocols used between the WSADMIN client and the Deployment Manager. The two are very similar, though RMI has certain benefits over SOAP. We'll explore RMI a little later.
- Security issues; specifically why using the WebSphere Admin ID makes things easier, and how we could use a different ID if we wished to. We'll cover those topics down under "Security Related Exercises" starting on page 85.

In this lesson we learned a few important things:

○ The WSADMIN client is a shell script. Using the WSADMIN function requires the `wsadmin.sh` shell script to be invoked.

○ There are two basic "modes" -- remote mode, which implies a network connection to the running server; and local mode, which means the `wsadmin.sh` shell script will update the configuration files directly. Remote mode is achieved with the parameter `-conntype SOAP` (or `RMI`), and local mode is achieved with `-conntype NONE`.

○ Our basic rule is this: if the server is up (DMGR or standalone server), use remote mode to connect to the server. Local mode is intended for situations where the server is down.

○ By default, WSADMIN expects the scripting language to be JACL. However, JACL is "deprecated," meaning that while JACL still works, the future is Jython. New script development should be in Jython, not JACL.

○ Telling WSADMIN to expect Jython is accomplished in one of two ways -- by passing in the `-lang jython` parameter at invocation, or by changing the default scripting language as specified in the `wsadmin.properties`. file.

Now that we've learned to invoke the `wsadmin.sh` client, we'll start using it. That's the next exercise.

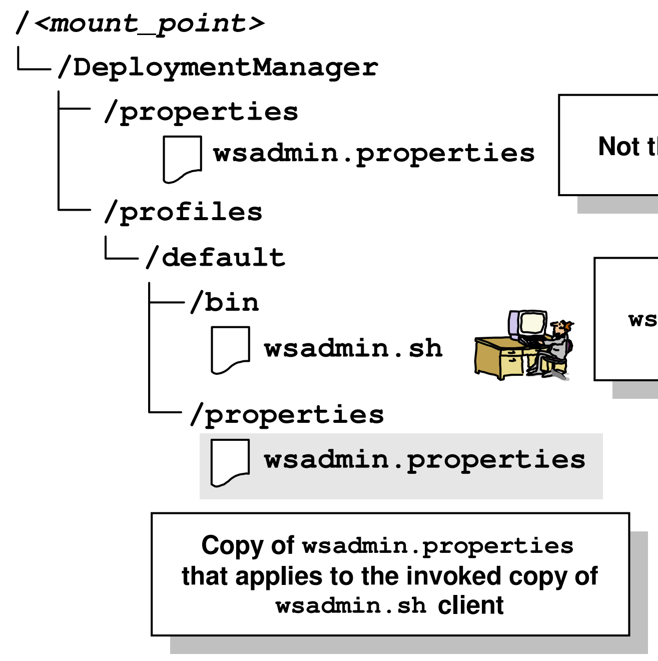
Command Line Usage and Jython Exercises

Lesson Overview

We'll change the default scripting language from JACL to Jython. We'll invoke WSADMIN using Jython and we'll start learning about the various objects. We'll start learning about some basic Jython things like variables.

Background: where the wsadmin.properties file is located

The `wsadmin.properties` file holds the property that defines the default script language for the WSADMIN client. There are two copies of this file in *each node's* HFS. The one that applies to us is the one under `/default/profiles/properties`.



Important: This file is in held in the HFS in **ASCII** format. Editing it on the MVS system requires tools to convert it to EBCDIC before tools like OEDIT will work. It's generally easier to FTP the file in *binary mode* down to the workstation, where an editor such as Notepad can be used. Then FTP it back to in binary mode.

But it's your choice ... whatever tool you know you can use to edit an ASCII file that's stored in the HFS will work. We'll show FTPing it to the workstation.

Exercise: change default script language of WSADMIN and verify

Objective	In this exercise you will modify the default language setting in the `wsadmin.properties` file. That will eliminate the need to code `-lang jython` on every WSADMIN invocation.

- ☐ Open up an FTP session to your host MVS
- ☐ Change directories to the `/DeploymentManager/profiles/default/properties` directory.
- ☐ Set the file transfer mode to binary.
- ☐ Download the file to your workstation.
- ☐ Edit the file and modify the line:

```
com.ibm.ws.scripting.defaultLang=jacl
```

changing the value `=jacl` to `=jython`. After you're done, it should look like this:

```
#---------------------------------------------------------------------------
# The defaultLang property determines what scripting language to use.
# Supported values are jacl and jython.
# The default value is jacl.
#---------------------------------------------------------------------------
com.ibm.ws.scripting.defaultLang=jython
```

☐ FTP the changed file back to the MVS system. Make sure you upload in binary mode.

☐ Check the permissions on the updated file in the HFS. Its permissions should be `644`.

☐ Now invoke WSADMIN from the Deployment Manager's `/profiles/default/bin` directory with the following command:

```
./wsadmin.sh -conntype SOAP -host aaaa -port bbbb -user cccc -password dddd
```

where:

- *aaaa* is the host address where the DMGR can be reached (see page 12)
- *bbbb* is the SOAP port of the DMGR (see page 12)
- *cccc* is the WebSphere Admin ID
- *dddd* is the password for the WebSphere Admin ID

You should see the following:

```
WASX7209I: Connected to process "dmgr" on node <node> using SOAP
connector;  The type of process is: DeploymentManager

WASX7031I: For help, enter: "print Help.help()"

wsadmin>
```

Note: The presence of the Jython format for the help command is evidence that the change you made was picked up. The default language is now Jython.

☐ Leave WSADMIN at the `wsadmin>` prompt.

Review That was a one-time change. The WSADMIN client invoked from the Deployment Manager's `/profiles/default/bin` directory will now read that `wsadmin.properties` file and assume a default scripting language of jython.

That wsadmin.properties file is also where the WSADMIN client picks up the host and port when `-host` and `-port` is not specified. As mentioned, we prefer to explicitly code those.

Exercise: simple application installation

Objective In this exercise you will install a very simple application called "SuperSnoop" into your Deployment Manager's master configuration.

Note: We will *not* use WSADMIN to synchronize the node. It *is* possible to do that, and we'll see that under "Exercise: node synchronization" on page 49. Here our objective is to install the application using the `AdminApp` object. But not actually use the application. That would require synchronization from the DMGR's master configuration out to the node.

☐ Go the Techdocs website and pull the SuperSnoop application EAR file:

```
http://www.ibm.com/support/techdocs/atsmastr.nsf/WebIndex/TD101815
```

☐ Place the `SuperSnoopProj.ear` file up on the z/OS in an HFS location that's accessible to the Deployment Manager server. The file should have permissions that permit of at least `644`. For example, place the file at:

```
/u/user1/SuperSnoopProj.ear
```

☐ Log onto the Admin Console and get the *long* name of the node and application server into which this application will be installed:

Node long name:	
Server long name:	

☐ At the `wsadmin>` prompt, enter the following command:

`AdminApp.install('/`*`aaaa`*`/SuperSnoopProj.ear','[-node `*`bbbb`*` -server `*`cccc`*`]')`

where:

- *`aaaa`* is location where you stored the EAR file
- *`bbbb`* is the long name of the node into which the application will be installed
- *`cccc`* is the long name of the application into which the application will be installed

Note: Where did we get the syntax of that? The InfoCenter offers helpful examples:

`http://publib.boulder.ibm.com/infocenter/wasinfo/v6r1/index.jsp?topic=/com.ibm.websphere.zseries.doc/info/zseries/ae/txml_callappinstall.html`

That long URL will take you straight to the page. Otherwise, you can go to the InfoCenter and search on something like `AdminApp.install` and start browsing.

We'll be showing more examples of syntax throughout this document.

If your command is properly constructed, you should see something like this:

```
ADMA5016I: Installation of SuperSnoop started.
ADMA5058I: Application and module versions are validated ...
  :
ADMA5011I: The cleanup of the temp dir for application SuperSnoop is complete.
ADMA5013I: Application SuperSnoop installed successfully.
```

☐ But the application is not yet saved. Do that with this command:

`AdminConfig.save()`

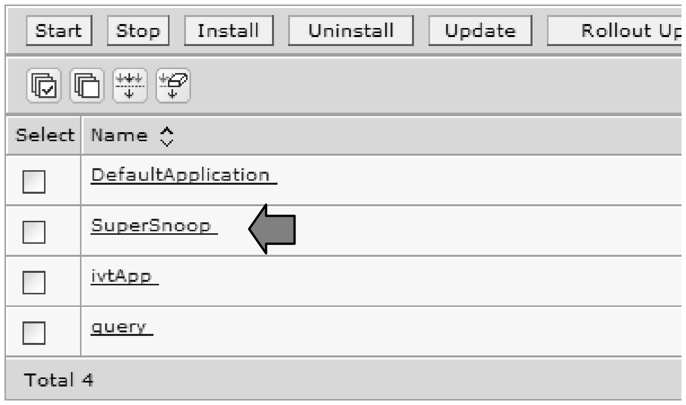
You won't see any positive confirmation of that. You'll simply be returned to the `wsadmin>` command prompt. You can verify the application is installed with this command:

`print AdminApp.list()`

You should see *something* like this:

```
DefaultApplication
SuperSnoop
ivtApp
query
wsadmin>
```

☐ Go the Admin Console and look at the applications installed in the cell:



Select	Name
☐	DefaultApplication
☐	SuperSnoop
☐	ivtApp
☐	query

Total 4

Review In this exercise you did several things:

- You changed the default script language from JACL to Jython
- You used the `AdminApp` object to install an application
- You used the `AdminApp` object to display a list of all applications currently in the cell

You are on your way to becoming a WSADMIN user.

Exercise: uninstall the application

Objective We'll turn around and uninstall the application.

☐ At the wsadmin> prompt enter:

```
AdminApp.uninstall('SuperSnoop')
```

You'll see something like this:

```
ADMA5017I: Uninstallation of SuperSnoop started.
ADMA5104I: The server index entry for WebSphere ... is updated successfully.
ADMA5102I: The configuration data for SuperSnoop ... is deleted successfully.
ADMA5011I: The cleanup of the temp dir for application SuperSnoop is complete.
ADMA5106I: Application SuperSnoop uninstalled successfully.
''
wsadmin>
```

☐ Issue the save command:

```
AdminConfig.save()
```

☐ Verify that the application is gone:

```
print AdminApp.list()
```

☐ You should see `SuperSnoop` now gone from the list.

Review More experience with the methods of `AdminApp`. So far you've used `install`, `list` and `uninstall`. If you turn back to page 13, you'll see a list of all the methods the AdminApp object has. You should see `install`, `list` and `uninstall` in that list.

Exercise: setting simple Jython variables

Objective We'll now start to explore the Jython language itself. What we'll do here has nothing to do with WSADMIN objects and methods. It's intended to showcase what you can do with the Jython scripting language.

When you do these exercise, keep in mind that ultimately we'll code Jython in a file and have WSADMIN read the file. Issuing the commands one after another at the command prompt is just a simple way to start showing you how things work.

Note: Much of what you'll see in the next few exercises came from the Techdoc:

`http://www.ibm.com/support/techdocs/atsmastr.nsf/WebIndex/`**`WP100963`**

written by Mike Loos. I would recommend you read that if you want a more thorough treatment of Jython. This white paper is taking some of that information and expanding it into a step-by-step "primer" style.

☐ Issue the command:

```
a = 1
```

☐ Now echo back the current value of the variable a:

```
print a
```

You should get back:

```
1
wsadmin>
```

☐ Issue the command

```
a = hello
```

You should get back

```
WASX7015E: Exception running command: "a = hello"; exception
information:
 com.ibm.bsf.BSFException: exception from Jython:
Traceback (innermost last):
  File "<input>", line 1, in ?
NameError: hello

wsadmin>
```

What that's saying is that string values can't be entered without some indication it's a string. That's done with quotes, which is next.

☐ Issue the command

```
a = 'hello'
```

Note: Double-quotes works equally well. But you can't start a string with a double quote and end with a single quote. The starting and ending quote must match.

☐ Now echo back the current value of the variable a:

```
print a
```

You should get back:

```
hello
wsadmin>
```

☐ Issue the following commands, one after another:

```
a = 1
b = 2
c = a + b
print c
```

You should get back:

```
3
wsadmin>
```

☐ Now try the following:

```
a = 1.11
b = 2.22
c = a * b
d = c ** 2
print d
```

You should get back:

```
6.072281640000002
wsadmin>
```

Note: That was $(1.11 \times 2.22)^2$

☐ Issue the following commands, one after another:

```
a = 'Hello '       ⇦ notice the space between the o and the single quote
b = 'World'
print a + b
```

You should get back:

```
Hello World
wsadmin>
```

☐ Another way to accomplish that would be:

```
a = 'Hello'        ⇦ notice no space this time
b = 'World'
print a , b
```

You should get back:

```
Hello World
wsadmin>
```

Note: The comma acted as a separator. Notice that Jython added a blank space between the two variables.

☐ Here's a bit more fancy string variable usage. This will "add" strings together to form a longer single string. Issue each command, one after another:

```
a = 'one'
b = 'two'
print 'The value of a is ' + a + ' and the value of b is '+ b
```

You should get back:

```
The value of a is one and the value of b is two
wsadmin>
```

Note: If your strings ran together without the blank spaces you desire, go back and look very carefully at how the blank spaces were specified *inside* the quotes.

☐ If you tried the last exercise when the value of either variable was numeric, it would throw an error. Here's how you mix string and numeric in a single print statement:

```
a = 1
b = 2
c = a + b
print 'The value of c is' , c
```

You should get back:

```
The value of c is 3
wsadmin>
```

Note: Again, the comma acted as a separator. Jython added a blank space between the string and the numeric variable value.

☐ Leave WSADMIN at the `wsadmin>` prompt.

Review A very simple set of exercises to allow you to see how numeric and string variables are set, acted upon (addition, multiplication) and printed back out.

Exercise: list variables

Objective	List variables are really element arrays. By that we mean the things in the variable are considered discrete -- that is, not simply part of a bigger string -- and can be parsed and used by the program.
	For example, the variable `a = [1, 2, 3, 4]` contains four discrete elements: the integers 1, 2, 3 and 4.
	We're going to focus on list variables because they play an important role in WSADMIN usage. Option lists for a WSADMIN object's method are passed in as a list. If you want to construct the list ahead of time as a variable, you must create a list variable.
	We saw an example of that earlier:
	`ear     = '/u/myapps/SuperSnoopProj.ear'`
	`node    = 'mynodec'`
	`server  = 'mysr01c'`
	`options = '[-node ' + node + ' -server ' + server + ']'`
	`AdminApp.install(ear,options)`
	`AdminConfig.save()`
	The options for the application install was created as a list variable called `options`. The variable was then used with the `AdminApp.install` method. That's a very common thing to do. Learning the essentials of list variables is a big part of mastering Jython for use with WSADMIN.

☐ Issue the following command:

```
options = [1, 2, 3, 4]
```

☐ Now Issue the following command

```
print options[2]
```

You should get back:

```
3
wsadmin>
```

Note: Why 3? Because lists have an starting offset of `0`. So the first element in the list is really accessed with offset `0`. Offset `2` is therefore the number 3.

☐ Now Issue the following command

```
options = options + [5]
```

Note: That appends the element 5 to the list. The list now consists of `[1, 2, 3, 4, 5]`.

☐ Test the appended element. Issue the command:

```
print options
```

You should get back:

```
[1, 2, 3, 4, 5]
wsadmin>
```

Note: Why are the square brackets part of that print options? It's Jython telling you the variable is really a list variable.

- ☐ Extract the last element:

```
last = options[4]
print last
```

You should get back:

```
5
wsadmin>
```

- ☐ Lists may also contain string variables. Do the following:

```
options = ['Hello', 'World']
print options
```

You should get back:

```
['Hello', 'World']
wsadmin>
```

- ☐ And of course you can extract strings as easily as numbers:

```
last = options[1]
print last
```

You should get back:

```
World
wsadmin>
```

- ☐ Numbers and strings can be mixed:

```
options = ['One', 2, 'Three', 4]
print options
```

You should get back:

```
['One', 2, 'Three', 4]
wsadmin>
```

- ☐ And to prove that elements `[1]` and `[3]` are really numbers:

```
x = options[1]
y = options[3]
z = x + y
print z
```

You should get back:

```
6
wsadmin>
```

- ☐ You can programmatically test for the length of a list variable:

```
a = len(options)
print a
```

You should get back:

```
4
wsadmin>
```

Note: The `len()` function returned the actual number of elements. It did *not* assume the first was counted as "0." So be careful with the output of `len()` and using the number to index into a list to extract a value.

☐ You can reset a list back to empty:

```
options = []
print options
```

You should get back:

```
[]
wsadmin>
```

☐ Finally, you can nest lists within lists:

```
opt1 = [1, 2, 3]
opt2 = ['eggs', 'milk', 'cheese']
opt3 = [opt1, opt2]
print opt3
```

You should get back:

```
[[1, 2, 3], ['eggs', 'milk', 'cheese']]
wsadmin>
```

Review List variables are special variables whose contents are discrete elements. This of it as an *array* of data, if that's helpful to you. You can test for the length of a list with the `len()` function, and you can parse out values using the index function `options[n]`, where n is the offset, starting with 0.

There's much more you can do with variables and list variables. Consult the Python tutorial at:

`http://www.python.org/doc/tut/tut.html`

Python was the predecessor to Jython. The syntax is nearly identical

Exercise: another way of constructing list variables

Objective To show how to create a list variable as a string. This is the more common way you'll see such list variables built for use by WSADMIN commands.

☐ At the WSADMIN prompt, issue the following, one after the other:

```
x = 'eggs'
y = 'milk'
z = 'cheese'
opt = '[' + x + ',' + y + ',' + z + ']'
print opt
```

You should get back:

```
[eggs,milk,cheese]
wsadmin>
```

Note: That is not truly a "list variable" right now ... Jython would consider that just a string of characters. But it would be what WSADMIN could use as an option list on a command. And you'll see us constructing option lists as strings quite frequently in this document.

Review Constructing lists as strings is a handy way of building WSADMIN options. Constructing lists in the manner we showed before is how you'd do it if you want to process the list programmatically, such as looping through it.

Lesson wrap-up and summary

Interactive mode is quite useful for small, ad hoc things. In this exercise we learned a few things about issuing commands from the command prompt, and a little about Jython itself.

File-based Input Exercises

Lesson Overview

Interactive mode is nice for small stuff, but beyond a few lines of Jython it gets tedious. Thankfully we're able to tell WSADMIN to read in a file with the Jython we wish to execute. File-based input allows us to code routines that execute the same way, time and time again.

In this unit we'll explore a number of different elements of WSADMIN and Jython:

- Using the `execfile()` function to pull the file while in interactive mode
- Specifying the file at WSADMIN invocation
- ASCII vs. EBCDIC encoding of the Jython file
- Passing parameters into a script
- Looping and if-then-else structures

Exercise: use the `execfile()` function

Objective	The `execfile()` function is a way to read in a Jython file from the WSADMIN command prompt. It's a way to have the best of both worlds: flexibility of interactive mode with the reusability of file-based Jython.

☐ If you still have the `wsadmin>` prompt, then continue. Otherwise, launch WSADMIN with the following command:

```
./wsadmin.sh –conntype SOAP –host aaaa –port bbbb –user cccc –password dddd
```

where:

- *aaaa* is the host address where the DMGR can be reached
- *bbbb* is the SOAP port of the DMGR
- *cccc* is the WebSphere Admin ID
- *dddd* is the password for the WebSphere Admin ID

Note: Remember that your telnet or OMVS session needs to be operating under that Admin ID before you issue the command to invoke WSADMIN.

☐ We need to get a file up on the z/OS system to read into WSADMIN. We'll do that by creating one in ASCII on workstation and FTP-ing it up to z/OS. Do the following:

☐ Open a Notepad session.

☐ Type the following into the Notepad session:

```
list_a = ['one', 'two', 'three', 'four', 'five']
for i in list_a:
  print 'The value of the item is', i
```

Note: Jython is sensitive to the column in which statements begin. Start each statement in column 1, except for things inside a code block, which is what follows the colon at end of `for` statement.

☐ Save the file as `test.jy`

☐ FTP the file to the z/OS system in *binary* mode and place in some directory such as `/u/user1` or whatever directory you have easy access to. Maintain the same name of `test.jy`.

Notes:
- We'll soon show you how to tell WSADMIN to accept EBCDIC input. For now, we need to make sure input is in ASCII because that's the default for WSADMIN.
- The file does *not* need to have an extension of `jy`. But it's as good as any.

- ☐ Make sure the permissions on that file are at least `644`

☐ Use the `execfile()` function to read in and execute that file. Issue the following command at the `wsadmin>` prompt:

```
execfile('/aaaa/test.jy')
```

where:

- *`aaaa`* is the directory location where you FTPed the file

You should get back:

```
The value of the item is one
The value of the item is two
The value of the item is three
The value of the item is four
The value of the item is five
wsadmin>
```

Review	You just executed a series of Jython commands -- in this case, the creation of a list variable and a simple loop that read through and printed out each element in the list -- and you executed them from a stored file. Coding Jython and WSADMIN in a file is a very important building block for using WSADMIN effectively. We'll now learn about a few other things having to do with executing WSADMIN from a stored file. In the next section we'll show how that's done from a batch JCL file.

Exercise: comments in the file (optional)

Objective	This exercise is so simple you may opt to skip it.

☐ Edit your workstation copy of `test.jy` and add the following comments:

```
list_a = ['one', 'two', 'three', 'four', 'five']
# Loop through and print
for i in list_a:
  # Printing each element
  print 'The value of the item is', i
```

☐ Save the file and FTP in binary mode up to the z/OS system.

☐ Invoke that file with the `execfile()` function, just as you did before. You should see the exact same results as before. The comments will not display.

Review	Once a script file gets beyond a few lines, comments will be needed to keep things orderly. We wanted to introduce how comments were coded.

Exercise: invoke WSADMIN and specifying input file

Objective	The `execfile()` function read in a file from the interactive command prompt, which meant that WSADMIN was already up. There's another way to read in a file -- at the time WSADMIN itself is invoked. If you have a WSADMIN script you want to run once and that's all you want to do, then having WSADMIN read the file in at invocation time, run the script and then exit may be just the thing for you.

☐ If you're still at the `wsadmin>` command prompt, enter **`quit`** to exit WSADMIN.

☐ Launch WSADMIN with the following command (all on one line):

Reminder: Compose in Notepad, then copy/paste into telnet or OMVS. Save the command string for use later.

```
./wsadmin.sh -conntype SOAP -lang jython -f /eeee/test.jy -host aaaa
                                    -port bbbb -user cccc -password dddd
```

where:

- `aaaa` is the host address where the DMGR can be reached
- `bbbb` is the SOAP port of the DMGR
- `cccc` is the WebSphere Admin ID
- `dddd` is the password for the WebSphere Admin ID
- `eeee` is the location where you stored the `test.jy` script file

Note: We saw that the `-lang` jython was necessary for WSADMIN to understand the language type, despite the default language being specified as jython in the `wsadmin.properties` file. Hence the inclusion of that parameter.

☐ Check to make sure you get back the following, which is exactly the same thing you saw when you used the `execfile()` function:

```
The value of the item is one
The value of the item is two
The value of the item is three
The value of the item is four
The value of the item is five
```

☐ Note that you are *not* at the `wsadmin>` prompt. You're at the UNIX prompt. That means that the WSADMIN client is *not* presently active.

The client came up, processed your input file, then closed back down.

Note: We hope you can see that it should be easy to issue the same invocation command from a JCL batch file. We'll cover that under "JCL and BPXBATCH Exercises" starting on page 34.

Review The ability to have WSADMIN come up, process a file, and then close back down positions us to do various WebSphere administrative things in batch mode.

Exercise: ASCII vs. EBCDIC encoding of Jython file

Objective The `test.jy` file in the z/OS system's HFS is presently in ASCII encoding. That's what WSADMIN expects by default. But you may want input in EBCDIC format, which would make it easier to create and edit using the standard z/OS editors. To do that, we need to tell WSADMIN to expect the EBCDIC. That's done with a somewhat strange Java option we pass into the WSADMIN client.

☐ On your workstation, copy the `test.jy` file to `test2.jy`.

☐ Edit the new `test2.jy` file and change it so it reads as follows:

```
list_a = ['this', 'file', 'is', 'in', 'EBCDIC']        ⇦ all changes this line
# Loop through and print
for i in list_a:
  # Printing each element
  print 'The value of the item is', i
```

☐ Save the file.

☐ FTP that file to the z/OS system, this time in **`ascii`** mode. That'll convert the file to EBCDIC format. Place in the same directory as the `test.jy` file was placed earlier. Make sure the file permissions are at least `644`.

☐ Using ISHELL or OMVS, use the standard z/OS editor to look at the file. You should be able to read that since the encoding is in EBCDIC.

Note: If the text is garbled, re-FTP and make sure the transfer mode is **`ascii`**, *not* binary.

☐ Launch WSADMIN with the following command (all on one line):

```
./wsadmin.sh -javaoption -Dscript.encoding=Cp1047 -conntype SOAP
                                    -lang jython -f /eeee/test2.jy -host aaaa
                                         -port bbbb -user cccc -password dddd
```

Note: Text shaded in gray is what's new compared to previous command issued. It's important the `-javaoption` come *before* the `-f` command. Parameters are processed in order seen, so in this case you'd want to tell WSADMIN about the encoding before it attempted to read in the file.

where:

- *`aaaa`* is the host address where the DMGR can be reached
- *`bbbb`* is the SOAP port of the DMGR
- *`cccc`* is the WebSphere Admin ID
- *`dddd`* is the password for the WebSphere Admin ID
- *`eeee`* is the location where you stored the `test2.jy` script file

☐ You should get back the following:

```
The value of the item is this
The value of the item is file
The value of the item is is
The value of the item is in
The value of the item is EBCDIC
```

Again, note that you are *not* at the `wsadmin>` prompt. You're at the UNIX prompt. That means that the WSADMIN client is *not* presently active. WSADMIN was invoked, the file specified with the `-f` option executed, and WSADMIN then closed down.

Review That is the key to having your input in EBCDIC encoding. You must tell WSADMIN to expect the non-default character encoding, and that's done with the `-javaoption` parameter to WSADMIN with `-Dscript.encoding=Cp1047` supplied.

Exercise: looping and other logic in Jython

Objective We've already one example of a "for" loop. Here we'll explore a few more. Most of this is coming straight from the Python tutorial at:

```
http://www.python.org/doc/tut/tut.html
```

They call these things "control flow tools."

We'll revert back to creating the script files on the workstation and FTPing them to the host in *binary* mode. That means the files will be stored on z/OS in ASCII, which is the default for WSADMIN. No `-javaoption` will be needed. We'll use `execfile()`.

☐ Your WSADMIN client should not be active. In case it is, close it now.

☐ Launch WSADMIN with the following command:

```
./wsadmin.sh -conntype SOAP -host aaaa -port bbbb -user cccc -password dddd
```

where:

- *`aaaa`* is the host address where the DMGR can be reached
- *`bbbb`* is the SOAP port of the DMGR
- *`cccc`* is the WebSphere Admin ID
- *`dddd`* is the password for the WebSphere Admin ID

The "if" and "elif" statements

☐ Create a file on your workstation with the name **`control1.jy`**

Note: At this point we're going to supply the exercise files in the ZIP file that is a companion to the PDF on Techdocs.

☐ Code the contents of file as:

```
x = 4
if x < 0:
  print 'The value of x is', x, 'which is less than zero'
elif x == 0:
  print 'The value of x is', x, 'which is exactly equal zero'
elif x > 0:
  print 'The value of x is', x, 'which is greater than zero'
```

☐ FTP the file in *binary* mode to the z/OS system. Make sure permissions are at least `644`.

☐ Invoke the file with:

```
execfile('/aaaa/control1.jy')
```

where *`aaaa`* is the location where you stored the file.

You should see:

```
The value of x is 4 which is greater than zero
wsadmin>
```

The "for" statement

☐ Create a file on your workstation with the name **`control2.jy`**

☐ Code the contents of file as:

```
count = 0
option = ['one', 'two', 'three', 'four', 'five', 1, 2, 3, 4, 5]
len_option = len(option)
print 'The length of the variable "option" is', len_option
for i in option:
  count = count + 1
  print 'Checking item', count, 'in the list. The value found is', i
print 'Done'
```

Notes:

- There may be a more elegant way to report the iteration count of the loop.
- Note the colon at the end of the `for` statement
- Note how the statements inside the for loop are *indented at least one space.* That's important ... that's what tells Jython that it's part of the loop
- The final `print 'Done'` statement is back at column 1

☐ FTP it to the z/OS system in binary. Make sure permissions are `644` minimum.

☐ Invoke with `execfile()` as you did before

☐ You should see back:

```
The length of the variable "option" is 10
Checking item 1 in the list. The value found is one
Checking item 2 in the list. The value found is two
Checking item 3 in the list. The value found is three
Checking item 4 in the list. The value found is four
Checking item 5 in the list. The value found is five
Checking item 6 in the list. The value found is 1
Checking item 7 in the list. The value found is 2
Checking item 8 in the list. The value found is 3
Checking item 9 in the list. The value found is 4
Checking item 10 in the list. The value found is 5
Done
```

The "break" statement

- ☐ Make of copy of `control2.jy` and call it **`control3.jy`**
- ☐ Make the following modifications. Code added is highlighted in gray:

```
count = 0
check = 'four'
option = ['one', 'two', 'three', 'four', 'five', 1, 2, 3, 4, 5]
len_option = len(option)
print 'The length of the variable "option" is', len_option
for i in option:
  count = count + 1
  print 'Checking item', count, 'in the list. The value found is', i
  if i == check:
    print 'Found item at position', count
    break
print 'Done'
```

Notes:
- Notice how the `if` statement has a colon at the end
- Notice how the following `print` statement is indented further
- The `break` statement breaks whatever loop processing was taking place when the statement encountered. In this case it was the `for i in option:` loop

- ☐ FTP it to the z/OS system in binary. Make sure permissions are `644` minimum.
- ☐ Invoke with `execfile()` as you did before
- ☐ You should see back:

```
The length of the variable "option" is 10
Checking item 1 in the list. The value found is one
Checking item 2 in the list. The value found is two
Checking item 3 in the list. The value found is three
Checking item 4 in the list. The value found is four
Found item at position 4
Done
wsadmin>
```

Review A quick review of the `if` statement, the `for` looping function an the `break` function. These things are not strictly required to invoke the methods on WSADMIN objects. But you'll find them handy when you start building slightly more intelligent Jython scripts. This is particularly true if you plan to do validation testing on parameters passed into the script.

Exercise: passing parameters into a script

Objective Imagine a script that installs an application into a server. We saw earlier that such a script requires the EAR file be specified, and the server and node be named as well. You could hard-code those values as variables at the top of the script. That certainly works. But you may want to code a script that is a bit more flexible.

What we'll explore now is how you can pass parameters into a script, and then programmatically extract the parameters and use them in your script processing.

To do this, we're going to leave the `execfile()` function and go back to invoking WSADMIN and using the `-f` parameter to point to the script file.

- ☐ If you still have a `wsadmin>` prompt, issue the **`quit`** command to close it.
- ☐ Create a file on your workstation with the name **`parms.jy`**
- ☐ Code the contents of file as:

```
import sys
len_args = len(sys.argv)
print 'The number of parms passed in:', len_args
for i in sys.argv:
  print 'Parm value:', i
print 'All done'
```

Notes:
- The special variable `sys.argv` is where the passed-in parameters will go. We need to tell Jython about that special variable. We do that by importing the Java `sys` package.
- You custom parameters will be added to the end of the `./wsadmin.sh` invocation statement, after the -f parameter that points to the script file.
- The rest of this script involves things we've already looked at.

☐ FTP it to the z/OS system in binary. Make sure permissions are `644` minimum.

☐ Launch WSADMIN with the following command (all on one line):

```
./wsadmin.sh -conntype SOAP -lang jython -host aaaa -port bbbb
                -user cccc -password dddd -f /eeee/parms.jy milk eggs cheese
```

where:

- *aaaa* is the host address where the DMGR can be reached
- *bbbb* is the SOAP port of the DMGR
- *cccc* is the WebSphere Admin ID
- *dddd* is the password for the WebSphere Admin ID
- *eeee* is the location where you stored the `test.jy` script file
- and the parameters follow ... in this example, `milk eggs cheese`

☐ You should see back:

```
WASX7303I: The following options are passed to the scripting
environment and are available as arguments that are stored in
the argv variable: "[milk, eggs, cheese]"
The number of parms passed in: 3
Parm value: milk
Parm value: eggs
Parm value: cheese
All done
```

Review	The special variable `sys.argv` is very useful when you want a script to be more generic, with specific information passed in at time of invocation. The example shown here is rather simple: it does no validity checking to make sure the correct parameters are passed in, or that they're provided in the proper sequence. There are ways to do that, and we'll some of that later.

Lesson wrap-up and summary

In this set of lessons you saw some fundamentally important things about coding Jython in scripts and having WSADMIN read those scripts in and process them. We saw that the file can be encoded in either ASCII or EBCDIC format, but if EBCDIC then you need to inform WSADMIN of that fact.

We saw some of the basic elements of Jython logical control. And we saw how to pass parameters into a script using the `sys.argv` variable.

You are becoming well positioned to delve deeper into the specifics of the WSADMIN objects, and how you can programmatically do things you've used the Admin Console for up to now. We want first to cover how to use JCL and BPXBATCH to invoke WSADMIN because that will set the stage for true MVS batch operations. Then we'll explore the WSADMIN objects and show many examples of how to do key WebSphere things.

JCL and BPXBATCH Exercises

Lesson Overview

This will be a relative short set of exercises because invoking WSADMIN from JCL is really little different than doing it from a telnet or OMVS session. That's why we spent some time on the manual invocation of WSADMIN and the use of the `-f` option to point to a script file.

Exercise: simple invocation of WSADMIN

Objective	To get the JCL working, with WSADMIN successfully invoked and a script file processed.

☐ In a JCL library data set of your choosing, provide the following JCL, substituting in as needed according to the notes below the JCL:

Note: This JCL is supplied as file `wsadmin.jcl` in the accompanying ZIP file.

```
=COLS> ----+----1----+----2----+----3----+----4----+----5----+
****** ***************************** Top of Data *************
//WSADMIN JOB (ACCTNO,ROOM),REGION=0M,
// USER=aaaa,PASSWORD=bbbb
//STEP1    EXEC PGM=IKJEFT01
//SYSTSPRT DD SYSOUT=*
//SYSTSIN  DD *
 BPXBATCH SH +
  /cccc/DeploymentManager+                      ⇦ Note: no space before + sign
  /profiles/default/bin/wsadmin.sh +            ⇦ Note: there is a space before + sign
  -lang jython +
  -conntype SOAP +
  -host ddd.ddd.ddd +
  -port eeee +
  -user aaaa +
  -password bbbb +
  -f /ffff/gggg +                               ⇦ Note: point to control3.jy used earlier
 1> /tmp/wsadmin.out +
 2> /tmp/wsadmin.err
/*
```

where:

- *aaaa* -- is the WebSphere Admin ID
- *bbbb* -- is the WebSphere Admin ID password
- *cccc* -- is the mount point for the DMGR node
- *ddd.ddd.ddd* -- is the host name where the DMGR is listening
- *eeee* -- is the SOAP port (or ORB port if `-conntype RMI` used)
- `ffff` -- is the directory location where the script file is located
- `gggg` -- is the name of the script file

Note: When there's no space before the + sign it means the concatenation is made with no spaces inserted. That's exactly what we want when we built the path to the `wsadmin.sh` shell script. But everywhere else we **do** want a blank space, hence the space before the + sign.

What we're building here is pretty much exactly like what we built when we invoked WSADMIN manually by entering all this in at the telnet prompt.

The `1>` and `2>` is there `STDOUT` and `STDERR` are going to go.

☐ Update the `JOB` card to you local requirements

☐ Update the `USER=` and `PASSWORD=` on the JOB card with the WebSphere Admin ID.

> **Note:** If you are not permitted to code a password in the JOB card, then make sure this job runs under the authority of the WebSphere Admin ID. That is important because this will attempt to establish a SOAP connection to the DMGR, and that will mean SSL if security is enabled. The WebSphere Admin ID will have the keyring and certificates; other IDs will not.
>
> See "Exercise: use a different ID from WebSphere Admin ID" on page 85 for more on running JCL batch under a different userid authority.

☐ Make sure the `-f` option points to the **`control3.jy`** file you created earlier.

☐ Submit this job.

☐ Look for `RC=0` in the `JESMSGLG` for the submitted job. You won't see the output from the script here.

☐ If `RC=0`, then look in the `/tmp/wsadmin.out` to see the output; if *other than* `RC=0` and not some simple JCL error, then look in the `/tmp/wsadmin.err` to see what might have gone wrong.

☐ If it worked, the output in `/tmp/wsadmin.out` should look like this:

```
******************************** Top of Data *********
WASX7209I: Connected to process "dmgr" on node <node>
The length of the variable "option" is 10
Checking item 1 in the list. The value found is one
Checking item 2 in the list. The value found is two
Checking item 3 in the list. The value found is three
Checking item 4 in the list. The value found is four
Found item at position 4
Done
******************************** Bottom of Data *******
```

Review The invocation of WSADMIN with a JCL batch file and BPXBATCH truly is just like issuing the command at a telnet command prompt.

Exercise: passing in parameters

Objective To see how parameters can be passed into a WSADMIN script when invoking from JCL. This is fairly simple: it involves the same JCL with the parameters added after the `-f` pointer to the script file. We'll change the script we run to `parms.jy`.

☐ Edit the WSADMIN JCL you created in the previous exercise. Change the Jython script file name being invoked from `control3.jy` to `parms.jy`.

> **Note:** Both script files are provided in the companion ZIP file to this PDF on Techdocs. See the top of this page for the Techdoc number.

☐ Now add parameters after the `parms.jy` file name but *before* the + sign. Be sure to leave a space between your last parameter and the + sign.

```
   :
   -f /u/user1/parms.jy eggs milk cheese +
  1> /tmp/wsadmin.out +
  2> /tmp/wsadmin.err
/*
```

☐ Submit the job and review the output as before.

☐ If it worked, the output in `/tmp/wsadmin.out` should look like this:

```
WASX7209I: Connected to process "dmgr" on node <node> using SOAP connector; The
type of process is: DeploymentManager
WASX7303I: The following options are passed to the scripting environment and are
available as arguments that are stored in the argv variable: "[eggs, milk, cheese]"
The number of parms passed in: 3
Parm value: eggs
Parm value: milk
Parm value: cheese
All done
```

Review Again, no different than when the command was entered at the telnet prompt.

Lesson wrap-up and summary

Using JCL as the tool to control the invocation of WSADMIN is a very useful thing. It can't be used for interactive ad hoc type things, but once you have a script developed and working, invoking it using batch JCL may help non-UNIX people use WSADMIN.

Hint: During script development you may wish to use the `execfile()` function from a WSADMIN command prompt. That saves the time of starting up WSADMIN each time you want to test some new portion of your script.

Exploring the WSADMIN Objects Exercises

Becoming an expert in WSADMIN

Is not an easy thing to do. Knowing that five WSADMIN "objects" exist is the easy part. Understanding all the methods on those objects and -- more difficult still -- all the options those methods have is what takes time.

With WebSphere V6.1 we have something that helps -- the "Command Assistance" function of the Admin Console. It will report on the WSADMIN command that is equivalent to some action taken in the console. But it's not perfect; do not expect it to provide you with all the information you may need.

WSADMIN is like any programming language: learning it involves starting out with some relatively simple things, then branching out from there. And making liberal use of reference material where available. This is where the InfoCenter comes into play. The InfoCenter is an *excellent* reference source for WSADMIN. So is the Internet ... you'd be surprised how many examples of WSADMIN scripts are starting to appear out there.

With all that in mind, what we're going to do now is wade into WSADMIN, starting out relatively slow with fairly simple things, then building up to more complicated things.

Topic: Admin Console command assistance

Lesson Overview

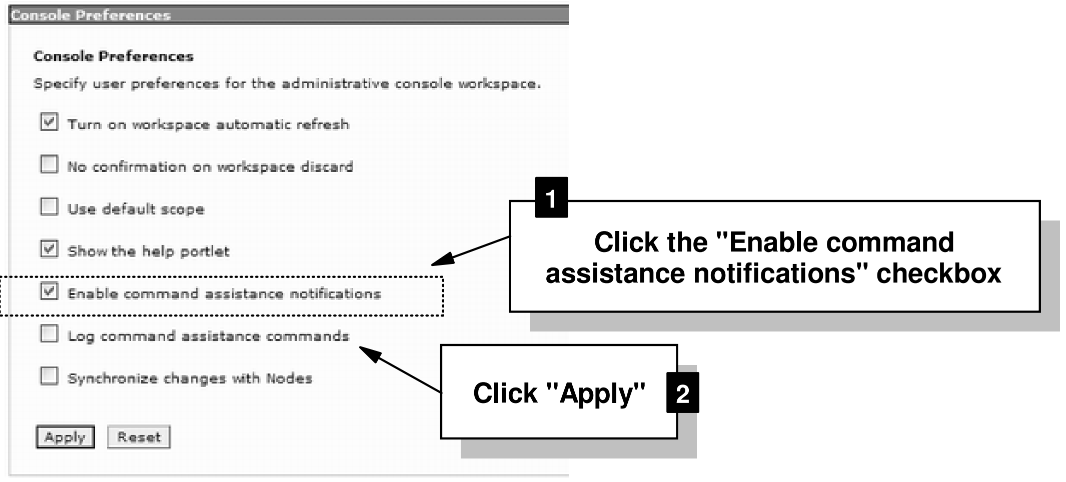
The new Command Assistance feature of WebSphere Application Server Version 6.1 is a feature of the Admin Console that will report the WSADMIN command equivalent to the action taken in the console. In this lesson you will learn how to enable and use the feature.

Exercise: enable command assistance in Admin Console

Objective	To see where in the console we have the ability to enable or disable the Command Assistance function, and how to quickly verify that it is working.

☐ In the Admin Console, go to ***System Administration*** ⇨ ***Console Preferences***

☐ Enable command assistance. Do the following:


Console Preferences

Console Preferences

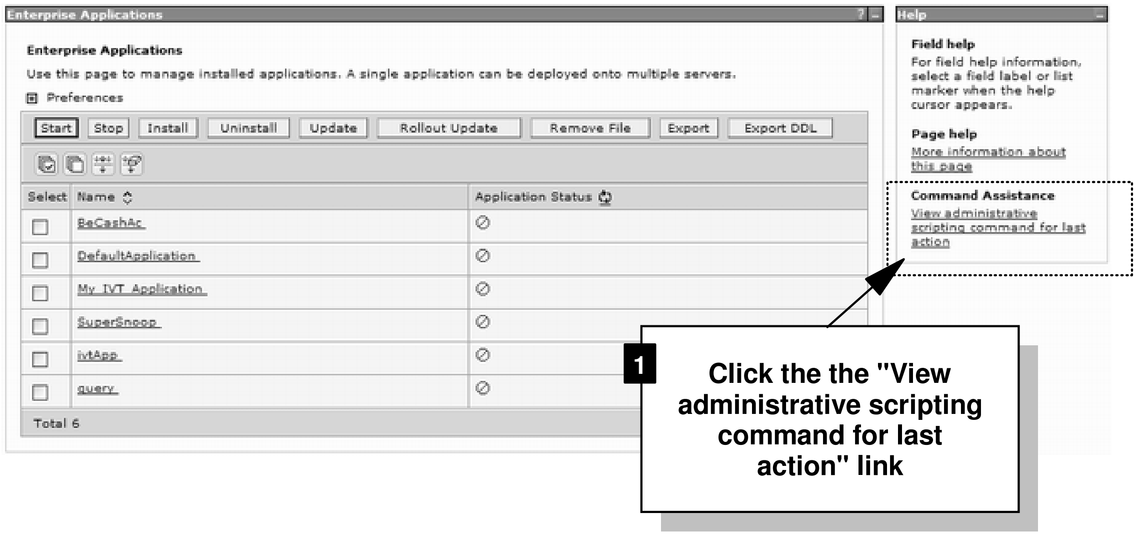
Specify user preferences for the administrative console workspace.

- [✓] Turn on workspace automatic refresh
- [ ] No confirmation on workspace discard
- [ ] Use default scope
- [✓] Show the help portlet
- [✓] Enable command assistance notifications
- [ ] Log command assistance commands
- [ ] Synchronize changes with Nodes

Apply
Reset
1
Click the "Enable command assistance notifications" checkbox
Click "Apply"
2

☐ In the Admin Console, go to ***Applications*** ⇨ ***Enterprise Applications***

☐ You should see a list of whatever applications you have installed, along with a link for "command assistance" over on the right side of the screen:



Enterprise Applications

Use this page to manage installed applications. A single application can be deployed onto multiple servers.

⊞ Preferences

Select	Name	Application Status
☐	BeCashAc	⊘
☐	DefaultApplication	⊘
☐	My IVT Application	⊘
☐	SuperSnoop	⊘
☐	ivtApp	⊘
☐	query	⊘

Total 6

Field help

For field help information, select a field label or list marker when the help cursor appears.

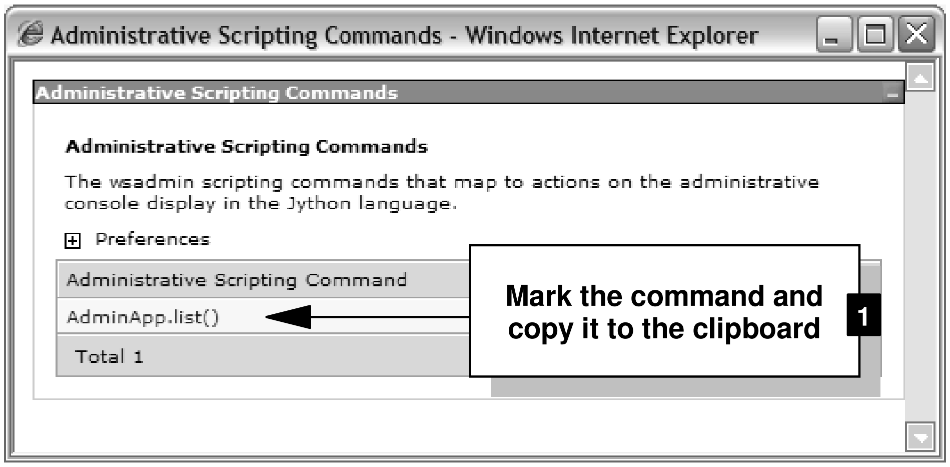
Page help

More information about this page

Command Assistance

View administrative scripting command for last action

☐ Using your mouse, copy the command out of the window that's provided:


Administrative Scripting Commands - Windows Internet Explorer
Mark the command and copy it to the clipboard
1

Administrative Scripting Commands

The wsadmin scripting commands that map to actions on the administrative console display in the Jython language.

⊞ Preferences

Administrative Scripting Command
AdminApp.list()

Total 1

Note: `AdminApp.list()` is the WSADMIN Jython command used to list the applications in the cell.

☐ Paste the command into a Notepad session, or any text editor. You have the very beginnings of a Jython script.

Review	A very simple exercise to validate that command assistance is turned on. We of course are interested in more complex commands than this. We will get to those in a moment.

Exercise: enable logging of command assistance

Objective	Displaying the commands to the Admin Console is one method of doing this; logging them to a file is another method. The file the command is written to is deep inside the configuration file system for the DMGR, and will be held in ASCII. You'll see where that file is in this exercise.

☐ In the Admin Console, go to ***System Administration*** ⇨ ***Console Preferences***

☐ Click the checkbox for "Log command assistance commands"

☐ Click on the "Apply" button.

- ☐ In the Admin Console, go to ***Applications*** ⇨ ***Enterprise Applications***. This will once again list all the applications installed in the cell.
- ☐ In a Telnet session, go to:

  `/`***`aaaa`***`/DeploymentManager/profiles/default/logs/dmgr`

  where ***`aaaa`*** is the mount point for your Deployment Manager's configuration file system.
- ☐ In that directory you should see a file named:

  `commandAssistanceJythonCommands_`**`aaaa`**`.log`

  where ***`aaaa`*** is the ID you used when you logged into the Admin Console.
- ☐ That file is in ASCII encoding. Open the file for browsing and scroll to the bottom. You should see something like the following:

```
# [4/24/07 13:03:47:560 GMT+00:00] ApplicationDeployment
AdminApp.list()
```

- ☐ Mark the command (*not* the date/timestamp) and copy to the clipboard, then paste to a separate Notepad session or other text editor session. Again, you've demonstrated how you can extract the command for your own scripting use.
- ☐ Close the "Command Assistance" window.

Review	You've seen another way to capture the WSADMIN Jython command equivalent to an Admin Console action.

Exercise: a more complex command

Objective	The `AdminApp.list()` example was used because it was an easy and non-disruptive command to generate. But our interest in the command assistance feature is that it provides a way to see far more complex commands; command structures we might not otherwise be able to figure out. Here we'll see an example of that: we'll create a new server and see the resulting command. We will *not* save the change, however. So in reality nothing gets changed in the WebSphere configuration. It will illustrate an important point: you can use the Admin Console to capture WSADMIN commands without disturbing the actual configuration. You simply avoid the final "save" process.

Note: This exercise assumes a Network Deployment configuration. It won't work for a Standalone.

- ☐ In the Admin Console, go to ***Servers*** ⇨ ***Application Servers***.
- ☐ Click the **New** button.
- ☐ Provide a "Server Name" of **`test`**.
- ☐ At "Server Template," click **Next**.
- ☐ For "Server specific short name" provide **`SPECIFIC`**
- ☐ For "Server generic short name" provide **`GENERIC`**
- ☐ Click **Next**
- ☐ At "Confirm New Server" click **Finish**
- ☐ You should see the "View administrative scripting command for last action" link to the right side of the screen. Click on that link.

☐ You should see the associated WSADMIN command:

```
AdminTask.createApplicationServer('aaaa', '[-name test
-templateName defaultZOS -specificShortName SPECIFIC
-genericShortName GENERIC ]')
```

where *aaaa* is the long name of the node where the server would be created.

Note: The `AdminTask` object is new with V6.1. It has some very useful functions that are now part of single command that used to take multiple commands before. The method `createApplicationServer()` is what creates the server. What's inside the `()` are the option for the method. This is where the "Command Assistance" feature is so helpful -- here it is giving us an example of the options we would need, as well as the syntax of those options.

☐ Close the "Command Assistance" window.

☐ Click the **Review** link at the top of the screen. This will give us a chance to discard the changes we've made.

☐ Click on **Discard** and then **Yes** so the new server is *not* created.

Review	The creation of a server is a more complex thing than simply listing out the applications that are installed. We saw that the command assistance feature will handle the more complex thing.

Lesson wrap-up and summary

The purpose of this exercise was to demonstrate how the new "Command Assistance" feature of WebSphere Application Server V6.1 can be used to get a good example of the WSADMIN command syntax for a given action.

Topic: use "help" to get a sense for what methods are on what objects

Lesson Overview

The online help facility does not really offer much along the lines of precise syntax charts. But it can be useful in determining what methods are applicable to what objects, and to a lesser degree the properties of the method itself. In this exercise we'll do just a little exploration of the "help" facility.

Exercise: display contents of `help()` method

Objective	To see that the online help can display what information is available.

☐ Get to a WSADMIN command prompt.

Note: The option `-conntype NONE` works well for simple help exploration.

☐ Issue the following command:

```
print Help.help()
```

You should get something like this (some preliminary output removed to save space):

```
attributes              given an MBean, returns help for attributes
operations              given an MBean, returns help for operations
constructors            given an MBean, returns help for constructors
description             given an MBean, returns help for description
notifications           given an MBean, returns help for notifications
classname               given an MBean, returns help for classname
all                     given an MBean, returns help for all the above
help                    returns this help text
AdminControl            returns general help text for the AdminControl object
AdminConfig             returns general help text for the AdminConfig object
```

```
AdminApp                returns general help text for the AdminApp object
AdminTask               returns general help text for the AdminTask object
wsadmin                 returns general help text for the wsadmin script
                        launcher
message                 given a message id, returns explanation and
                        user action message
```

The ones in bold are what we'll explore now.

☐ Issue the following command:

`print AdminControl.help()`

You should get something like this (some preliminary output removed to save space):

```
completeObjectName
                Return a String version of an object name given a
                template name
getAttribute_jmx
                Given ObjectName and name of attribute, returns value of
                attribute
getAttribute    Given String version of ObjectName and name of attribute,
                returns value of attribute
getAttributes_jmx
                Given ObjectName and array of attribute names, returns
                AttributeList
getAttributes   Given String version of ObjectName and attribute names,
                returns String of name value pairs

getCell         returns the cell name of the connected server
getConfigId     Given String version of ObjectName, return a config id for
                the corresponding configuration object, if any.
getDefaultDomain
                returns "WebSphere"
getDomainName   returns "WebSphere"

getHost         returns String representation of connected host
getMBeanCount   returns number of registered beans
getMBeanInfo_jmx
                Given ObjectName, returns MBeanInfo structure for MBean

getNode         returns the node name of the connected server
getObjectInstance
                Given String version of ObjectName, returns
                ObjectInstance object that match.
getPort         returns String representation of port in use
getType         returns String representation of connection type in use
help            Show help information
invoke_jmx      Given ObjectName, name of method, array of parameters and
                signature, invoke method on MBean specified
invoke          Invoke a method on the specified MBean
isRegistered_jmx
                true if supplied ObjectName is registered
isRegistered    true if supplied String version of ObjectName is registered
makeObjectName  Return an ObjectName built with the given string
queryNames_jmx  Given ObjectName and QueryExp, retrieves set of ObjectNames
                that match.
queryNames      Given String version of ObjectName, retrieves String of
                ObjectNames that match.
queryMBeans     Given String version of ObjectName, returns a set of
                ObjectInstances object that match.
reconnect       reconnects with server
setAttribute_jmx
                Given ObjectName and Attribute object, set attribute for MBean
                specified
setAttribute    Given String version of ObjectName, attribute name and
                attribute value, set attribute for MBean specified
setAttributes_jmx
                Given ObjectName and AttributeList object, set attributes for
                the MBean specified
setAttributes   Given String version of ObjectName, attribute name
```

```
                and value pairs, set attributes for the MBean specified
startServer     Given the name of a server, start that server.
stopServer      Given the name of a server, stop that server.
testConnection  Test the connection to a DataSource object
trace           Set the wsadmin trace specification
```

Note: Those are the various things you can do with the `AdminControl` object. We'll show how the help facility can be used to drill down for more information. That's next.

☐ Let's pick the `getCell` method and see what the format of that command is. Issue the following command:

```
print AdminControl.help('getCell')
```

You should see:

```
WASX7332I: Method: getCell

        Arguments: none
        Description: Returns the cell to which the scripting process is
        connected.
```

Note: That particular method has no arguments. That means the format of the command would be:

```
AdminControl.getCell()
```

That would return the cell long name. You could put that into a variable if you wished:

```
cell_long = AdminControl.getCell()
```

The `help()` method of any object can be used to provide more information on any one of the other methods.

☐ Now let's look at the `help()` method on the other objects. Issue the following commands, one after the other:

```
print AdminConfig.help()
```

```
print AdminApp.help()
```

```
print AdminTask.help()
```

You should see:

AdminConfig

```
attributes      Show the attributes for a given type
checkin         Check a file into the the config repository.
convertToCluster
                converts a server to be the first member of a
                new ServerCluster
create          Creates a configuration object, given a type, a parent, and
                a list of attributes, and optionally an attribute name for the
                new object
createClusterMember
                Creates a new server that is a member of an
                existing cluster.
createDocument  Creates a new document in the config repository.
createUsingTemplate
                Creates an object using a particular template type.
defaults        Displays the default values for attributes of a given type.
deleteDocument  Deletes a document from the config repository.
existsDocument  Tests for the existence of a document in the config repository.
extract         Extract a file from the config repository.
getCrossDocumentValidationEnabled
                Returns true if cross-document validation is enabled.
getid           Show the configId of an object, given a string version of
                its containment
getObjectName   Given a config id, return a string version of the ObjectName
                for the corresponding running MBean, if any.
```

```
getSaveMode     Returns the mode used when "save" is invoked
getValidationLevel
                Returns the validation used when files are extracted from the
                repository.
getValidationSeverityResult
                Returns the number of messages of a given
                severity from the most recent validation.
hasChanges      Returns true if unsaved configuration changes exist
help            Show help information
installResourceAdapter
                Installs a J2C resource adapter with the given rar
                file name and an option string in the node.
list            Lists all configuration objects of a given type
listTemplates   Lists all available configuration templates of a given
                type.
modify          Change specified attributes of a given configuration object
parents         Show the objects which contain a given type
queryChanges    Returns a list of unsaved files
remove          Removes the specified configuration object
required        Displays the required attributes of a given type.
reset           Discard unsaved configuration changes
save            Commit unsaved changes to the configuration repository
setCrossDocumentValidationEnabled
                Sets the cross-document validation enabled mode.
setSaveMode     Changes the mode used when "save" is invoked
setValidationLevel
                Sets the validation used when files are extracted from the
                repository.
show            Show the attributes of a given configuration object
showall         Recursively show the attributes of a given configuration
                object, and all the objects contained within each attribute.
showAttribute   Displays only the value for the single attribute specified.
types           Show the possible types for configuration
uninstallResourceAdapter
                Uninstalls a J2C resource adapter with the given
                resource adapter config id.
validate        Invokes validation
```

AdminApp

```
deleteUserAndGroupEntries
                Deletes all the user/group information for all
                the roles and all the username/password information for RunAs
                roles for a given application.
edit            Edit the properties of an application
editInteractive Edit the properties of an application interactively
export          Export application to a file
exportDDL       Export DDL from application to a directory
getDeployStatus Returns the combined Deployment status of the application
help            Show help information

install         Installs an application, given a file name and an option string.

installInteractive
                Installs an application in interactive mode, given a
                file name and an option string.
isAppReady      Checks whether the application is ready to be run
list            List all installed applications
listModules     List the modules in a specified application
options         Shows the options available, for a given file, application,
                or in general.
publishWSDL     Publish WSDL files for a given application
searchJNDIReferences
                List application that refer to the given JNDIName on a given
node
taskInfo        Shows detailed information pertaining to a given install task
                for a given file
uninstall       Uninstalls an application, given an application name and
                an option string
update          Updates an installed application
updateAccessIDs Updates the user/group binding information with accessID
```

```
                from user registry for a given application
updateInteractive       Updates an installed application interactively
view            View an application or module,
                given an application or module name
```

AdminTask

```
help -commands                  list all the admin commands
help -commandGroups             list all the admin command groups
help commandName                display detailed information for
                                the specified command
help commandName stepName       display detailed information for
                                the specified step belonging to the
                                specified command
help commandGroupName           display detailed information for
                                the specified command group

There are various flavors to invoke an admin command. They are

commandName                     invokes an admin command that does not require
                                any argument.

commandName targetObject        invokes an admin command with the specified
                                target object string, for example, the
                                configuration object name of a resource
                                adapter. The expected target object varies
                                with the admin command invoked. Use help
                                command to get information on the target
                                object of an admin command.

commandName options             invokes an admin command with the specified
                                option strings. This invocation syntax is
                                used to invoke an admin command that does
                                not require a target object. It is also
                                used to enter interactive mode if
                                "-interactive" mode is included in the
                                options string.

commandName targetObject options        invokes an admin command with the
                                        specified target object and options
strings.
                                        If "-interactive" is included in the
options
                                        string, then interactive mode is
entered.
                                        The target object and options strings
                                        vary depending on the admin command
invoked.
                                        Use help command to get information
                                        on the target object and options.
```

Review	That's a lot of information. Do not expect to know all the specifics of all those methods right away. Most you will probably never use. We invoked the `help()` method so you could see a listing of all the possible methods on the different objects.

Exercise: drill down on the `AdminApp` object

Objective	To show that the online help can be used to explore deeper into commands.

☐ Issue the following command:

```
print AdminApp.help('install')
```

This will give some information on the `install()` method. You should see:

```
WASX7096I: Method: install

        Arguments: filename, options

        Description: Installs the application in the file specified by
        "filename" using the options specified by "options". All required
```

```
information must be supplied in the options string; no prompting is
performed.

The AdminApp "options" command may be used to get a list of all
possible options for a given ear file.  The AdminApp "help" command
may be used to get more information about each particular option.
```

Note: In other words, the command syntax is:

`AdminApp.install('`*`filename`*`', '`*`options`*`')`

The *filename* is easy -- it would be something like `'/u/user1/myapp.ear'`. The *options* component is what's challenging. What options you supply depends on what you're looking to do -- not all the possible options are required each time -- and what's in the EAR file being installed. Next we'll see how we can learn a little more about the possible options.

☐ Earlier (page 19) you put the `SuperSnoopProj.ear` file in the HFS and installed it with the AdminApp.install command. Locate where you placed that EAR file. You'll need that location for these exercises.

☐ Issue the following command:

`print AdminApp.options('/`*`aaaa`*`/SuperSnoopProj.ear')`

where *`aaaa`* is the location where the `SuperSnoopProj.ear` is located in the HFS. This will report back all the `AdminApp` options that are applicable to the EAR file. You should see something like this:

```
WASX7112I: The following options are valid for "/aaaa/SuperSnoopProj.ear"
MapModulesToServers
MapWebModToVH
CtxRootForWebMod
MapSharedLibForMod
JSPCompileOptions
JSPReloadForWebMod
GetServerName
preCompileJSPs
nopreCompileJSPs
distributeApp
nodistributeApp
useMetaDataFromBinary
nouseMetaDataFromBinary
deployejb
nodeployejb
createMBeansForResources
nocreateMBeansForResources
reloadEnabled
noreloadEnabled
deployws
nodeployws
processEmbeddedConfig
noprocessEmbeddedConfig
allowDispatchRemoteInclude
noallowDispatchRemoteInclude
allowServiceRemoteInclude
noallowServiceRemoteInclude
usedefaultbindings
defaultbinding.force
allowPermInFilterPolicy
noallowPermInFilterPolicy
verbose
update
update.ignore.old
update.ignore.new
installed.ear.destination
appname
reloadInterval
validateinstall
```

```
filepermission
buildVersion
deployejb.rmic
deployejb.dbtype
deployejb.dbschema
deployejb.classpath
deployws.classpath
deployws.jardirs
defaultbinding.datasource.jndi
defaultbinding.datasource.username
defaultbinding.datasource.password
defaultbinding.cf.jndi
defaultbinding.cf.resauth
defaultbinding.ejbjndi.prefix
defaultbinding.virtual.host
defaultbinding.strategy.file
filepermission
target
server
node
cell
cluster
contextroot
custom
installed.ear.destination
```

Note: Three important things to note:

1. That list is *not* necessarily the entire list of options available; it simply represents the options that are applicable to that particular EAR file.
2. All those are **not** required. This is simply a listing of what is possible given the EAR file that was pointed to.
3. There is a nice summary table of all these in the InfoCenter:

   `http://publib.boulder.ibm.com/infocenter/wasinfo/v6r1/index.jsp?topic=/com.ibm.websphere.zseries.doc/info/zseries/ae/rxml_taskoptions.html`

☐ Issue the following command:

```
print AdminApp.taskInfo('/aaaa/SuperSnoopProj.ear','CtxRootForWebMod')
```

where *aaaa* is the location where the EAR file. This will give you specific information on the usage of the `CtxRootForWebMod` option for the specified EAR file.

Note: The "context root" is the value on the URL that follows the host and port. It is what tells WebSphere which web module to invoke. The default context root for the SuperSnoop application is `/SuperSnoopWeb`. But you can change that at deployment time using the `CtxRootForWebMod` option on the `AdminApp.install()` command.

You should see something like this:

```
CtxRootForWebMod: Edit the Context root of web module

Context root defined in the deployment descriptor can be edited.

WASX7348I: ADMINAPP_TASKINFO=WASX7348I: Each element of the CtxRootForWebMod
task consists of the following 3 fields: "Web module", "URI", "ContextRoot".
Of these fields, the following are required and used as keys to locate the rows
in the task: "Web module" "URI" and the following may be assigned new values:

The current contents of the task after running default bindings are:
Web module: SuperSnoopWeb  1
URI: SuperSnoopWeb.war,WEB-INF/web.xml 2
ContextRoot: SuperSnoopWeb  3
```

Note: A little explanation is in order:

1. The `taskInfo` method is reporting back what it saw in the EAR file; namely, a single web module with a name of `SuperSnoopWeb`. An EAR file may have multiple web modules; this would report all it saw.
2. The "URI" is this output refers to a kind of unique identifier for the web module within the EAR file. When an EAR file contains multiple web modules, it's important that any WSADMIN script that looks to modify the properties of a web module points precisely to the one to be changed. The "URI" is how a web module is uniquely specified.
3. This is the current context root for the web module, as defined in the `web.xml` descriptor file inside the EAR.

Review We saw that the `help()` method can be useful for understanding the usage of some of the options. In particular, it can be used to interrogate an EAR and report on what it sees in the EAR.

Lesson wrap-up and summary

Understanding the exact syntax of the options is perhaps the most challenging aspect of learning WSADMIN. The `help()` function assists with this, as does the command assistance as does the InfoCenter. So does a lot of testing and trial and error.

Topic: WSADMIN and application-oriented activities

Note: The usage of fancy Jython techniques will be held to a minimum here. The objective is to show the basics of WSADMIN usage. Once you learn that you can expand and wrapper it with more complex Jython logic if you wish. The techdoc at:

`http://www.ibm.com/support/techdocs/atsmastr.nsf/WebIndex/`**`WP100963`**

has many good examples of more advanced Jython scripting.

Lesson Overview

We'll cover some common application-oriented tasks. This will make use of the AdminApp object mostly, but others as needed to complete the given task. We're going to use an application called `MyIVT.ear`, which we supply in the ZIP file that accompanies the techdoc on `ibm.com/support/techdocs`. MyIVT is a simple application consisting of a single servlet, a JSP and a stateless session bean. It requires no connections to external data sources, so it makes a nice application for simple testing and verification purposes.

Exercise: simple installation and listing of installed applications

Objective This will illustrate the most basic useful task -- installing an application programmatically.

- ☐ Start a telnet session. Switch users to your WebSphere Admin ID.
- ☐ Open a WSADMIN command prompt session connecting to your DMGR using the `-conntype SOAP` or `-conntype RMI` option.
- ☐ Issue the command:

```
print AdminApp.list()
```

  This will produce a listing of currently installed applications.
- ☐ Create a file on your workstation called `app1.jy` and add the following to it:

Note: The file `app1.jy` is supplied as part of the ZIP file that accompanies the techdoc on `ibm.com/support/techdocs`. So too is the `MyIVT.ear` file.

```
server  = 'aaaa'
node    = 'bbbb'
ear     = '/cccc/MyIVT.ear'
options = '[-node ' + node + ' -server ' + server + ']'
AdminApp.install(ear,options)
AdminConfig.save()
print AdminApp.list()
```

where:

- `aaaa` -- is the long name of the server into which the application will be installed
- `bbbb` -- is the long name of the node where the server resides
- `cccc` -- is the location in the z/OS file system where the EAR file will be placed

Note: We saw this earlier in the document. A very simple construction of three string variables and a list variable, then the invocation of the `AdminApp` and `AdminConfig` objects.

☐ FTP the `MyIVT.ear` file to the z/OS system in *binary* format. Place it in the same file system location as you specified in the Jython script file. Make sure permissions permit the file to be read ... `644` at a minimum.

☐ FTP the `app1.jy` file to the z/OS system in *binary* format. Make sure permissions permit the file to be read ... `644` at a minimum.

Note: You may store the Jython script file in EBCDIC on the z/OS system if you wish. You must then make sure your WSADMIN session is invoked with the -javaoption as shown back under "Exercise: ASCII vs. EBCDIC encoding of Jython file" on page 29. That would make editing the file on z/OS easier.

For this document we're going to assume you compose your Jython scripts on the workstation and FTP them to the host in binary so they're stored there in ASCII. By default that's what WSADMIN expects.

☐ Invoke that Jython file with the following command:

```
execfile('/aaaa/app1.jy')
```

where `aaaa` is the location in the z/OS file system where you stored the file.

You should see *something* like this:

```
ADMA5016I: Installation of My_IVT_Application started.
ADMA5058I: Application and module versions are validated ...
ADMA5005I: The application My_IVT_Application is configured ...
ADMA5053I: The library references for the installed optional package created.
ADMA5005I: The application My_IVT_Application is configured ...
ADMA5001I: The application binaries are saved in ...
ADMA5005I: The application My_IVT_Application is configured ...
SECJ0400I: Successfuly updated the application My_IVT_Application ...
ADMA5011I: The cleanup of the temp directory for application is complete.
ADMA5013I: Application My_IVT_Application installed successfully.
BeCashAc
DefaultApplication
My_IVT_Application
ivtApp
query
wsadmin>
```

Note: The application is installed and saved, but *not* synchronized and *not* started. So it can't be used in its current state. We'll see how to do that next.

Review The script just shown is the basic application installation skeleton. You can use that as a starting point and build it up with more Jython to do logic-related tasks if you wish.

Exercise: node synchronization

Objective	To show how to invoke node synchronization using WSADMIN.

Note: Node synchronization (and starting/stopping applications) is a funny thing. It's not a simple command. It involves something called a "management bean" -- a Java object that's part of WebSphere and commonly known by the short name "mBean." The act of synchronization involves specifying the appropriate mBean and invoking a function on that bean. For synchronization, the mBean is on the Node Agent. So we need to uniquely point to the mBean on the Node Agent for the node we want to synchronize, then reach out and touch that mBean and invoke the `sync` function.

Try to keep that in mind as you work through this exercise.

☐ The first thing we'll explore is something called the `completeObjectname` method on the `AdminControl` object. This returns the unique name of the synchronization mBean object in the node we wish to synchronize. *This does not synchronize the node*. This simply captures the unique mBean name ... the "complete object name."

Issue the following command at the WSADMIN prompt:

```
x = AdminControl.completeObjectName('type=NodeSync,node=aaaa,*')
```

where *`aaaa`* is the long name of the node you wish to synchronize. This will place the complete object name into the variable `x`.

Note: What we're *really* doing there is use a kind wildcard search with *just enough* specified so we can be assured to get a single hit on the mBean we need. By providing `type=NodeSync` we narrow it down to just `NodeSync` mBeans. By specifying the node long name we narrow it down further to just the `NodeSync` mBeans for a given node. For any node there's only one `NodeSync` mBean. Therefore we get a single hit, and the complete object name is placed into the variable.

There will be no response to this ... you will simple get the WSADMIN prompt back.

☐ Now print out the contents of the variable `x`:

```
print x
```

You should see *something* like this, with *your* cell and node names, not mine:

```
WebSphere:name=nodeSync,process=nodeagent,platform=common,node=fznodec,d
iagnosticProvider=true,version=6.1.0.4,type=NodeSync,mbeanIdentifier=nod
eSync,cell=fzcell,spec=1.0
```

Notes:

- That is the unique name -- the `completeObjectName` -- of the `nodeSync` mBean on the Node Agent. This is what we need to programmatically "touch" and invoke the `sync` function. We'll do that next.
- What it have been possible to skip the programmatic retrieval of the complete object name and just hard code the value? Yes, that is possible. But that's awkward and somewhat inflexible.

☐ With the node's `NodeSync` mBean complete object name stored in the variable `x`, we may now invoke node synchronization. Issue the following command:

```
AdminControl.invoke(x,'sync')
```

When synchronization is complete WSADMIN will return `'true'`. If that takes a short period of time it means the node is already synchronized or the changes are relatively small and synchronization completed quickly. If it takes a little longer, it means more needed to be synchronized between the DMGR and the node.

☐ Now let's combine all that into a single routine. Create a file on your workstation called `app2.jy` and add the following to it:

```
myNodeName = 'aaaa'
beanName = AdminControl.completeObjectName('type=NodeSync,node=' + myNodeName + ',*')
AdminControl.invoke(beanName,'sync')
```

where *`aaaa`* is the long name of the node being synchronized.

☐ FTP that file to the z/OS file system in binary node and make sure the permissions allow WSADMIN to access it.

☐ Invoke that Jython file with the following command:

```
execfile('/aaaa/app2.jy')
```

where *`aaaa`* is the location in the z/OS file system where you stored the file. You won't see anything in response ... you'll just get the WSADMIN prompt back.

Note: This synchronization routine is *not* perfect. It's missing two things:

1. It is for a single node and not all the nodes in the cell. It is possible to programmatically get a list of the nodes in a cell and loop through, synchronizing them all. We'll see that next.
2. It assumes the node agent is up so synchronization can actually take place. If the node agent is not up the process will throw an error: `WASX7025E: Error found in String ""; cannot create ObjectName`. We'll see how to work around that next.

☐ Let's now see how to programmatically loop through all the nodes in a cell, synchronizing each in turn. Create a file on your workstation called `app3.jy` and add the following to it:

Note: The file `app3.jy` is supplied as part of the ZIP file that accompanies the techdoc on `ibm.com/support/techdocs`.

```
1  import java.lang.System as sys
   lineSeparator = sys.getProperty('line.separator')
   node_ids = AdminConfig.list('Node').split(lineSeparator) 2
3  for node in node_ids:
     node_name = AdminConfig.showAttribute(node,'name') <-- 4
     nodeSync = AdminControl.completeObjectName('type=NodeSync,node='\ 5
       + node_name + ',*')
 6-> if nodeSync != "":
       AdminControl.invoke(nodeSync,'sync') 7
```

There's much to explain with this sample. Notes:

1. The first two lines are housekeeping so we can use a make use of a Java utility to reformat what the `AdminConfig.list()` command will return. See the next note.
2. Here we use the `AdminConfig.list()` command to get a list of all the nodes in the cell. We extend the command with `.split(lineSeparator)`, which we prepared for in the first two lines. Without this the list return would be in one big blob; with this it separates each node name out into a list variable we can then loop through. The list variable we place this in is called `node_ids`. That's what we'll loop through next.
3. We enter our loop to process through the list of nodes. For each iteration of the loop, the variable `node` is populated with the next node name from the list variable.
4. The `AdminConfig.showAttribute()` command is used to transform the node name returned from `AdminConfig.list()` into what we normally think of as the "node long

name." We do that because the next command we use here requires the traditional long name format, not the format returned by `AdminConfig.list()`.

5. We now get the "complete object name" for the first node in the list.. We need this because the `AdminControl.invoke()` command, used to invoke synchronization, requires that complete object name format. The backward slash at the end is a way to break a line and continue it on the next line, as shown.
6. Now we check if the value returned from `AdminControl.getCompleteObjectname()` is *not* null. If it *is* null, it means one of two things:
   a. The node is a Deployment Manager node. We do not want to attempt to synchronize to a DMGR node, or
   b. The node agent for the node is not up. If no node agent, then synchronization can't occur, so we don't even try
7. If the complete object name is not null, then we issue the `AdminControl.invoke()` command to synchronize with the node.

☐ FTP that file to the z/OS file system in binary node and make sure the permissions allow WSADMIN to access it.

☐ Invoke that Jython file with the following command:

`execfile('/`*`aaaa`*`/app3.jy')`

where *`aaaa`* is the location in the z/OS file system where you stored the file. You won't see anything in response ... you'll just get the WSADMIN prompt back.

Note: You now have a nice generic synchronization routine that will programmatically discover and synchronize all the nodes in your cell. You can call this Jython routine from another Jython file using the `execfile()` call. So if you have a WSADMIN script that requires synchronization, you can either put this synchronization routine into the script, or just call it using `execfile()`.

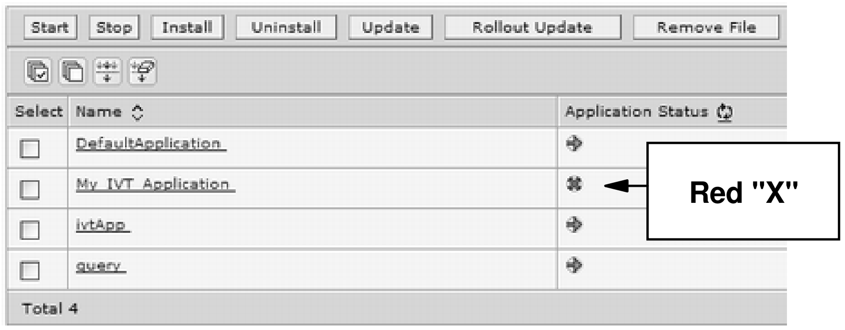
Review Node synchronization involves programmatically touching the mBean of the Node Agent and invoking the `sync` function. You can hard-code the way the Node Agent mBean is located and invoked, or programmatically get a list of nodes and loop through that list.

Exercise: starting and stopping an application

Objective Installing an application is one step, but before it can be used it has to be started. This can be done programmatically using WSADMIN.

Note: This exercise assumes the application server is up and running.

☐ Go to the Admin Console and look at the status of your applications. You should see MyIVT there, but in a stopped state:



Select	Name	Application Status
☐	DefaultApplication	➜
☐	My IVT Application	✖
☐	ivtApp	➜
☐	query	➜

Total 4

☐ Create a file on your workstation called `app4.jy` and add the following to it:

```
1  app_srv  = 'aaaa'
   app_name = 'bbbb'
2  app_mgr  = AdminControl.queryNames('type=ApplicationManager,process='\
      + app_srv + ',*')
   AdminControl.invoke(app_mgr,'startApplication',app_name)
                      3
```

Notes:

1. Where *aaaa* is the long name of the server in which the application is deployed, and *bbbb* is the display name of the application (for MyIVT, it would be `My_IVT_Application`). These values are placed into variables.
2. The `AdminControl.queryNames()` command is used to extract the unique name of the `ApplicationManager` mBean in the server and place it into the variable `app_mgr`. The line was continued with the backslash, and the variable `app_srv`, set earlier, was used to substitute in the server name.
3. The `AdminControl.invoke()` command was used to start the application. That takes three attributes: the application manager mBean name (which is very long and complicated, which is why we have it in a variable), the function you wish to invoke (`startApplication` in this example) and the name of the application being started.

☐ FTP that file to the z/OS file system in binary node and make sure the permissions allow WSADMIN to access it.

☐ Invoke that Jython file with the following command:

```
execfile('/aaaa/app4.jy')
```

where *aaaa* is the location in the z/OS file system where you stored the file. You won't see anything in response ... you'll just get the WSADMIN prompt back.

☐ Go back to the Admin Console and check the status of the application. You may need to click on the little "double-arrow" icon to the right of "Application Status" to refresh the screen so the green arrow shows.

Notes:
- If you have access to the SYSOUT of the servant region you can also see the message "Application started".
- The application may now be used. The URL is:

  `http://<your_server>:<port>/MyIVT/index.html`

  Instructions for using the application is on that front HTML page.

☐ To stop an application is very similar. Copy `app4.jy` to `app5.jy`. Then make the following changes:

```
app_srv  = 'aaaa' 1
app_name = 'bbbb'
app_mgr  = AdminControl.queryNames('type=ApplicationManager,process='\
   + app_srv + ',*')
AdminControl.invoke(app_mgr,'stopApplication',app_name)
                             2
```

Notes:

1. Where *aaaa* is the long name of the server in which the application is deployed, and *bbbb* is the display name of the application.

2. The string `start` is changed to `stop`.

- ☐ FTP that file to the z/OS file system in binary node and make sure the permissions allow WSADMIN to access it.
- ☐ Invoke that Jython file with the following command:

```
execfile('/aaaa/app5.jy')
```

where *aaaa* is the location in the z/OS file system where you stored the file. You won't see anything in response ... you'll just get the WSADMIN prompt back.

- ☐ Go back to the Admin Console and check the status of the application. You may need to click on the little "double-arrow" icon to the right of "Application Status" to refresh the screen so the red X shows.

Note: If you have access to the `SYSOUT` of the servant region you can also see the message "Application stopped".

Review Like synchronization, the act of starting and stopping an application involved identifying the management bean in the server and invoking a function on that bean.

Exercise: uninstall application

Objective To show how an application is uninstalled. We're going to combine a few things together here to show how more and more of this process can be automated.

- ☐ Create a file on your workstation called `app6.jy` and add the following to it:

Note: The file `app6.jy` (and all exercise scripts that are shown in this document) is supplied as part of the ZIP file that accompanies the techdoc on `ibm.com/support/techdocs`.

```
  # ---------------------------------------------------------------
1 # Set variables
  # ---------------------------------------------------------------
  app_srv  =    'aaaa'
  app_name =    'bbbb' 2
  sync_script = 'cccc'
  # ---------------------------------------------------------------
  # Stop application
  # ---------------------------------------------------------------
  print 'Stopping application', app_name
  app_mgr  = AdminControl.queryNames('type=ApplicationManager,process='\
3    + app_srv + ',*')
  AdminControl.invoke(app_mgr,'stopApplication',app_name)
  print 'Application', app_name, 'stopped'
  # ---------------------------------------------------------------
  # Uninstall application
  # ---------------------------------------------------------------
  print 'Uninstalling application', app_name
  AdminApp.uninstall(app_name)
4 AdminConfig.save()
  print 'Application', app_name, 'uninstalled'
  # ---------------------------------------------------------------
  # Invoking synchronization
  # ---------------------------------------------------------------
  print 'Invoking node synchronization'
5 execfile(sync_script)
  print 'Synchronization complete'
```

Notes:

1. We're showing the use of comments to make the scripts cleaner and easier to read.
2. Where *aaaa* is the long name of the server in which the application is deployed, *bbbb* is the display name of the application, and *cccc* is the location and Jython file name of the script that does node synchronization. In our exercises here, that was `app3.jy`. Remember, that was generic enough to be used as a callable service for any script needing node synchronization.
3. This is the same code we showed before to stop an application. If the application is already down this code will be executed but have no real effect. We could programmatically check for the status of the application, but doing this is easier and just as effective.
4. The actual uninstall code. We used the `AdminApp.uninstall()` command and provide the name of the application ... here in the form of a variable we set at the top of the script. We do not need to tell WebSphere what server it is in. Just the application name is all that's needed.

Note: This code example assumes the application exists in the application repository. If it wasn't there, then you'd see the error `WASX7280E: An application with name "My_IVT_Application" does not exist`. The script would end right there.

If you wanted to programmatically check to see if the application was present before attempting the uninstall, you could use `AdminApp.list()` and place the output into a list variable. Then loop through the list checking for the application name. We won't show that because that's really just more of the same kind of Jython processing we showed earlier.

5. The use of `execfile()` to call the node synchronization script we created earlier. With this we can start to see that WSADMIN scripts can be written as reusable modules.

☐ FTP that file to the z/OS file system in binary node and make sure the permissions allow WSADMIN to access it.

☐ Invoke that Jython file with the following command:

`execfile('/`*`aaaa`*`/app6.jy')`

where *aaaa* is the location in the z/OS file system where you stored the file. You see the following:

```
Stopping application My_IVT_Application
Application My_IVT_Application stopped
Uninstalling application My_IVT_Application
ADMA5017I: Uninstallation of My_IVT_Application started.
ADMA5104I: The server index entry for WebSphere:cell=fzcell,node=fznodec
is updated successfully.
ADMA5102I: The configuration data for My_IVT_Application from the
configuration repository is deleted successfully.
ADMA5011I: The cleanup of the temp directory for application
My_IVT_Application is complete.
ADMA5106I: Application My_IVT_Application uninstalled successfully.
Application My_IVT_Application uninstalled
Invoking node synchronization
Synchronization complete
wsadmin>
```

Review Uninstalling an application is quite simple with the `AdminApp.uninstall()` function. We added more to that to show a more robust scripting example.

Exercise: combined installation, synchronization and application starting in one script

Objective	This is really a variation on the previous script. The order of the components inside the script is a slightly different.

☐ Create a file on your workstation called `app7.jy` and add the following to it:

Note: The file `app7.jy` (and all exercise scripts that are shown in this document) is supplied as part of the ZIP file that accompanies the techdoc on `ibm.com/support/techdocs`.

```
  # --------------------------------------------------------------------
  # Set variables
  # --------------------------------------------------------------------
  app_srv     =  'aaaa'
  app_node    =  'bbbb'
  app_ear     =  'cccc' 1
  app_name    =  'dddd'
  sync_script =  'eeee'
  # --------------------------------------------------------------------
  # Install application
  # --------------------------------------------------------------------
  print 'Installing application', app_name
2 options = '[-node ' + app_node + ' -server ' + app_srv + ']'
  AdminApp.install(app_ear,options)
  AdminConfig.save()
  print AdminApp.list()
  # --------------------------------------------------------------------
  # Invoking synchronization
  # --------------------------------------------------------------------
  print 'Invoking node synchronization'
3 execfile(sync_script)
  print 'Synchronization complete'
  # --------------------------------------------------------------------
  # Start application
  # --------------------------------------------------------------------
  print 'Starting application', app_name
  app_mgr  = AdminControl.queryNames('type=ApplicationManager,process='\
4    + app_srv + ',*')
  AdminControl.invoke(app_mgr,'startApplication',app_name)
  print 'Application', app_name, 'started'
```

Notes:

1. Where:
   - *`aaaa`* -- is the long name of the server into which the application will be installed
   - *`bbbb`* -- is the long name of the node where the server resides
   - *`cccc`* -- is the location and file name of where the EAR file is stored in the HFS
   - *`dddd`* -- is the "display name" of the application itself. You can see what that is by looking in the `application.xml` file, which is in the root of the EAR file. Use some zip tool to open the EAR and browse the application.xml file. Look for the `<display-name>` tag. We need this to start the application.
   - *`eeee`* -- is the location and script name of the node synchronization script you've developed. In the earlier exercise it was `app3.jy`.
2. The installation of the application using `AdminApp.install()`
3. The invocation of the node synchronization script
4. The starting of the application using `AdminControl.invoke()`

- ☐ FTP that file to the z/OS file system in binary node and make sure the permissions allow WSADMIN to access it.
- ☐ Invoke that Jython file with the following command:

  execfile('/*aaaa*/app7.jy')

  where *aaaa* is the location in the z/OS file system where you stored the file. You see the something like the following:

```
Installing application My_IVT_Application
ADMA5016I: Installation of My_IVT_Application started.
 :
ADMA5013I: Application My_IVT_Application installed successfully.
DefaultApplication
My_IVT_Application
ivtApp
query
Invoking node synchronization
Synchronization complete
Starting application My_IVT_Application
Application My_IVT_Application started
wsadmin>
```

Review	This exercise showed how WSADMIN is like a set of building blocks. The install script was very much like the uninstall script, just in a slightly different order.

Background: `AdminApp.install()` options syntax and how to learn what to use

When you install an application through the Admin Console, you get the opportunity to do things like change the application name, change the "context root" on web modules, and change JNDI names for EJBs and JNDI references to those EJBs. You can do the same thing in WSADMIN, but the command syntax requires that you know the module names and something called the module "URI" (which is not the same thing as the URI used by browsers).

The values for those things is not intuitively obvious. And the application developer may or many not know what you're referring to. But the `AdminApp.taskInfo()` function can query an EAR file and report back the contents, giving you an inventory of those values in preparation for the development of your WSADMIN script.

The bad news is that the `AdminApp.taskInfo()`takes two parameters: the EAR file (that's easy), and the "taskName". What's the value for "taskName" that's related to mapping a JNDI name to an EJB? The interactive help doesn't say.

But this is where the Admin Console's "command assistance" feature comes in very handy. If you go through the motions of installing an application EAR by hand, modifying those application attributes you want to code in your WSADMIN script, the command assistant feature will provide a fairly clear idea of what those "taskName" values are.

We did that with the `MyIVT.ear` file. We did a "dummy" install of the application and modified five things: the application name, the web module's context root, the web module's virtual host mapping, the JDNI name applied to the EJB module, and the web module's JNDI reference to the EJB. We provided names that we could easily pick out, such as `New_App_Name`, and `New_Context_Root`. What the command assistance feature provided was this:

```
AdminApp.install('/u/user1/MyIVT.ear', '[
-nopreCompileJSPs
-distributeApp
-nouseMetaDataFromBinary
-nodeployejb
-appname New_App_Name 1
-createMBeansForResources
-reloadEnabled
-reloadInterval 3
-nodeployws
-validateinstall warn
-noprocessEmbeddedConfig
-filepermission .*\.dll=755#.*\.so=755#.*\.a=755#.*\.sl=755
-noallowDispatchRemoteInclude
-noallowServiceRemoteInclude
-BindJndiForEJBNonMessageBinding [[ "My IVT EJB Module Display Name"
  My_IVT_EJB_Name MyIVTStatelessSession.jar,META-INF/ejb-jar.xml
2 ejb/New_JNDI_Name ]]
-MapEJBRefToEJB [[ My_IVT_Webapp_Display_Name ""
  MyIVTWebApp.war,WEB-INF/web.xml ejb/ivtEJBObject
  com.ibm.websphere.ivt.ivtEJB.ivtEJBObject ejb/New_JNDI_Name ]] 3
-MapWebModToVH [[ My_IVT_Webapp_Display_Name
  MyIVTWebApp.war,WEB-INF/web.xml proxy_host ]] 4
-CtxRootForWebMod [[ My_IVT_Webapp_Display_Name
  MyIVTWebApp.war,WEB-INF/web.xml /New_Context_Root ]] 5
]' )
```

Note: The actual output from the command assistant feature was just a long blob of information. We've formatted that slightly here to make it more clear.

Where:

1. Shows the option used when a new application name is provided.
2. Shows the option used when a new JNDI name is provided to a EJB module
3. Shows the option used when a web module references an EJB module's JNDI name
4. Shows the option used when a web module is bound to a non-default virtual host
5. Shows the option used when a web module is given a new context root

Note: You *could* take that output and modify it for the installation of `MyIVT.ear`. But the specific information in there -- such as `MyIVTWebApp.war` -- is unique to *just that EAR file*. What we want to do is determine the appropriate values for any EAR file you're given. That's why we're going to explore the use of `AdminApp.taskInfo()` next ... it'll allow us to query an EAR and extract in a nice handy way the information we need to do some common things at application installation time.

Background: syntax for our highlighted `AdminApp.install()` options

When looking at the output from the command assistant it's natural to wonder what all that stuff is. The syntax for `-appname` is easy. But if we look at the option for the new context root, we see:

```
-CtxRootForWebMod [
  [ My_IVT_Webapp_Display_Name
    MyIVTWebApp.war,WEB-INF/web.xml
    /New_Context_Root ]
]
```

Note: Again, the proper format is all on one line. We broke it across several lines to highlight that the option has three attributes.

So, what are those attributes? Here's where the `AdminApp.taskInfo()` function comes in handy. For the `-CtxRootForWebMod` option `taskInfo()` -- which we'll see how to use next -- provided this:

```
The current contents of the task after running default bindings are:
Web module: My_IVT_Webapp_Display_Name
URI: MyIVTWebApp.war,WEB-INF/web.xml
ContextRoot: /MyIVT
```

And there's our answer. The first attribute is the "Web module," the second is the "URI," and the third is the "ContextRoot."

We now know the syntax of the `-CtxRootForWebMod` option:

```
-CtxRootForWebMod [[ aaaa bbbb cccc ]]
```

where *aaaa* is the web module name, *bbbb* is the URI and *cccc* is the context root value.

We come back to square one -- how do we know what the "web module" and "URI" values are for an EAR file we've been given? We use the `AdminApp.taskInfo()` function. That's next.

Exercise: using `taskInfo()` to get information on contents of application EAR file

Objective	To see how the `AdminApp.taskInfo()` function can be useful to get information about the contents of an EAR file.

☐ At the WSADMIN command prompt, issue the following command:

```
print AdminApp.taskInfo('/aaaa/MyIVT.ear','BindJndiForEJBNonMessageBinding')
```

where *aaaa* is the location in the z/OS file system where you stored.

This provides information for the JNDI name applied to the EJB. You should see this:

```
The current contents of the task after running default bindings are:
EJB module: My IVT EJB Module Display Name
EJB: My_IVT_EJB_Name
URI: MyIVTStatelessSession.jar,META-INF/ejb-jar.xml
Target Resource JNDI Name: ejb/My_IVT_Session_Bean_JNDI_Name
```

☐ At the WSADMIN command prompt, issue the following command:

```
print AdminApp.taskInfo('/aaaa/MyIVT.ear','MapEJBRefToEJB')
```

where *aaaa* is the location in the z/OS file system where you stored.

This provides information on the reference to the EJB JNDI from the web module. You should see this:

```
The current contents of the task after running default bindings are:
Module: My_IVT_Webapp_Display_Name
EJB:
URI: MyIVTWebApp.war,WEB-INF/web.xml
Resource Reference: ejb/ivtEJBObject
Class: com.ibm.websphere.ivt.ivtEJB.ivtEJBObject
Target Resource JNDI Name: ejb/My_IVT_Session_Bean_JNDI_Name
```

Note: See how "EJB" is null? When it comes to formatting the `-MapEJBRefToEJB` option the attributes are positionally dependent. Therefore we have to account for that null. We'll do so with `""`. Just be aware of those kinds of things.

☐ At the WSADMIN command prompt, issue the following command:

```
print AdminApp.taskInfo('/aaaa/MyIVT.ear','MapWebModToVH')
```

where *aaaa* is the location in the z/OS file system where you stored.

This provides information on mapping the web module to a virtual host. You should see this:

```
The current contents of the task after running default bindings are:
Web module: My_IVT_Webapp_Display_Name
URI: MyIVTWebApp.war,WEB-INF/web.xml
Virtual host: default_host
```

☐ At the WSADMIN command prompt, issue the following command:

```
print AdminApp.taskInfo('/aaaa/MyIVT.ear','CtxRootForWebMod')
```

where *aaaa* is the location in the z/OS file system where you stored.

This provides information on specifying the context root. You should see this:

```
The current contents of the task after running default bindings are:
Web module: My_IVT_Webapp_Display_Name
URI: MyIVTWebApp.war,WEB-INF/web.xml
ContextRoot: /MyIVT
```

Review The purpose of this exercise was to cull from the `MyIVT.ear` file the current values for things like "Web Module" and "URI" so we could format up the proper syntax for the `AdminApp.install()` command. If we want to change the context root, for example, we need to tell WSADMIN *which* web module to modify. MyIVT has only one web module, but an EAR file might have many. That's the purpose of that attributes -- to uniquely identify the component within the EAR that is to be modified at installation time.

Exercise: change the application name and context root at installation time

Objective We'll illustrate this process by changing two of the easier things: the application name and the context root for the web module.

☐ Uninstall the MyIVT application from the cell. You may do this with the Admin Console, or the WSADMIN uninstall script you created earlier. Be sure that you save the changes and synchronize.

Note: If you uninstall using the Admin Console, you may need to exit and reestablish your WSADMIN prompt. There's some caching that goes on. A way to work around this is to issue the `AdminConfig.save()` command from the WSADMIN prompt. That may flush the cache and allow you to proceed without reestablishing the WSADMIN prompt.

☐ Create a file on your workstation called `app8.jy` and add the following to it:

Note: The file `app8.jy` (and all exercise scripts that are shown in this document) is supplied as part of the ZIP file that accompanies the techdoc on `ibm.com/support/techdocs`.

```
  # --------------------------------------------------------------------
  # Set basic variables
  # --------------------------------------------------------------------
  app_srv     =  'aaaa'
  app_node    =  'bbbb'
  app_ear     =  'cccc' 1
  app_name    =  'dddd'
  sync_script =  'eeee'
  # --------------------------------------------------------------------
  # Set context root variables
  # --------------------------------------------------------------------
  web_mod     =  'ffff'
  web_uri     =  'gggg' 2
  web_ctx     =  'hhhh'
  # --------------------------------------------------------------------
  # Build interior context root option list
  # --------------------------------------------------------------------
  ctx_opt     =  '[[ ' + web_mod + ' ' + web_uri + ' ' + web_ctx + ' ]]'
3 print 'Value of context root interior option list:',ctx_opt
  # --------------------------------------------------------------------
  # Build AdminApp.install() option list
  # --------------------------------------------------------------------
  options = '[-node ' + app_node +\
   ' -server ' + app_srv +\
4  ' -appname ' + app_name +\
   ' -CtxRootForWebMod ' + ctx_opt + ']'
  print 'Value of overall option list:',options
  # --------------------------------------------------------------------
  # Install application
  # --------------------------------------------------------------------
  print 'Installing application', app_name
  AdminApp.install(app_ear,options)
  AdminConfig.save()
  print AdminApp.list()
  # --------------------------------------------------------------------
  # Invoking synchronization
  # --------------------------------------------------------------------
5 print 'Invoking node synchronization'
  execfile(sync_script)
  print 'Synchronization complete'
  # --------------------------------------------------------------------
  # Start application
  # --------------------------------------------------------------------
  print 'Starting application', app_name
  app_mgr  = AdminControl.queryNames('type=ApplicationManager,process='\
     + app_srv + ',*')
  AdminControl.invoke(app_mgr,'startApplication',app_name)
  print 'Application', app_name, 'started'
```

Notes:

1. Where these are the same basic variables as used before.
2. Where these are the values we discovered for the `-CtxRootForWebMod` task. `web_mod` is the "Web Module"; `web_uri` the URI value; and `web_ctx` the context root you wish to apply to the application. For example, the `MyIVT.ear` application would require:

```
# ----------------------------------------------------------------------
# Set context root variables
# ----------------------------------------------------------------------
web_mod     =  'My_IVT_Webapp_Display_Name'
web_uri     =  'MyIVTWebApp.war,WEB-INF/web.xml'
web_ctx     =  '/My_New_IVT'
```

Note: You *could* build the options variable all at once. But when list variables are in use, it's often best to build the interior list variables first, then piece together the outer structure second. It's just a little less confusing.

2. Here we build the option list for the `-CtxRootForWebMod` task. It follows the syntax we discovered from the command assistance dummy run we did earlier. The double square brackets is not a mistake -- it requires two to open and two to close.
3. We build the overall option list, just as we did before. But this time we embed the "interior option list" (the one for `-CtxRootForWebMod` option) using the `ctx_opt` variable.
4. The rest is identical to the installation script we created before.

☐ FTP that file to the z/OS file system in binary node and make sure the permissions allow WSADMIN to access it.

☐ Invoke that Jython file with the following command:

```
execfile('/aaaa/app8.jy')
```

Where *aaaa* is the location of your script file. You should see output similar to the earlier installation and uninstallation scripts.

Review With this exercise we're starting to see some of the more complex syntax structures of WSADMIN. The `-CtxRootForWebMod` option has its own options, which means we really had *two* option lists, one imbedded in the other.

Working out the syntax of these options is the challenge of WSADMIN. The command assistance function helps. But ultimately it involves some hunting around and some trial and error to get it to work.

Exercise: change JNDI name relationships at installation time

Objective This is very similar to what we just did, only with a different option. Rather than `-CtxRootForWebMod`, we're going use `-BindJndiForEJBNonMessageBinding` (that applies the JNDI name to the EJB module in `MyIVT.ear`) and the option `-MapEJBRefToEJB`, which is what provides the web module the JNDI pointer to the EJB.

We'll drop the `-CtxRootForWebMod` and `-appname` updates we made in the previous exercise. That's just to keep the exercise from getting too big.

☐ Uninstall the MyIVT application from the cell. You may do this with the Admin Console, or the WSADMIN uninstall script you created earlier. Be sure that you save the changes and synchronize.

Note: If you uninstall using the Admin Console, you may need to exit and reestablish your WSADMIN prompt. There's some caching that goes on. A way to work around this is to issue the `AdminConfig.save()` command from the WSADMIN prompt. That may flush the cache and allow you to proceed without reestablishing the WSADMIN prompt.

☐ Create a file on your workstation called `app9.jy` and add the following to it:

Note: The file `app9.jy` (and all exercise scripts that are shown in this document) is supplied as part of the ZIP file that accompanies the techdoc on `ibm.com/support/techdocs`.

```
# ----------------------------------------------------------------------
# Set basic variables
# ----------------------------------------------------------------------
app_srv     =  'aaaa'
app_node    =  'bbbb'
app_ear     =  'cccc'
app_name    =  'dddd'     1
sync_script =  'eeee'
# ----------------------------------------------------------------------
# Set variables for -BindJndiForEJBNonMessageBinding
# ----------------------------------------------------------------------
ejb_mod     =  '"My IVT EJB Module Display Name"'
2 ejb       =  'My_IVT_EJB_Name'
ejb_uri     =  'MyIVTStatelessSession.jar,META-INF/ejb-jar.xml'
ejb_jndi    =  'ejb/New_JNDI_Name'
# ----------------------------------------------------------------------
# Set variables for -MapEJBRefToEJB
# ----------------------------------------------------------------------
web_mod       =  'My_IVT_Webapp_Display_Name'
web_ejb       =  '""'
3 web_uri     =  'MyIVTWebApp.war,WEB-INF/web.xml'
web_res_ref   =  'ejb/ivtEJBObject'
web_class_ref =  'com.ibm.websphere.ivt.ivtEJB.ivtEJBObject'
ejb_jndi_ref  =  ejb_jndi
# ----------------------------------------------------------------------
# Build -BindJndiForEJBNonMessageBinding option list
# ----------------------------------------------------------------------
4 ejb_jndi_opt  =  '[[ ' + ejb_mod + ' ' + ejb +\
  ' ' + ejb_uri + ' ' + ejb_jndi + ' ]]'
print 'Value of ejb_jndi_opt:',ejb_jndi_opt
# ----------------------------------------------------------------------
# Build -MapEJBRefToEJB option list
# ----------------------------------------------------------------------
5 web_jndi_ref_opt  =  '[[ ' + web_mod + ' ' + web_ejb +\
 ' ' + web_uri + ' ' + web_res_ref +\
 ' ' + web_class_ref + ' ' + ejb_jndi_ref + ' ]]'
print 'Value of web_jndi_ref_opt option list:',web_jndi_ref_opt
# ----------------------------------------------------------------------
# Build AdminApp.install() option list
# ----------------------------------------------------------------------
6 options = '[-node ' + app_node +\
 ' -server ' + app_srv +\
 ' -BindJndiForEJBNonMessageBinding ' + ejb_jndi_opt +\
 ' -MapEJBRefToEJB ' + web_jndi_ref_opt + ']'
print 'Value of overall option list:',options
# ----------------------------------------------------------------------
# Install application
# ----------------------------------------------------------------------
print 'Installing application', app_name
AdminApp.install(app_ear,options)
AdminConfig.save()
print AdminApp.list()
# ----------------------------------------------------------------------
7 # Invoking synchronization
# ----------------------------------------------------------------------
print 'Invoking node synchronization'
execfile(sync_script)
print 'Synchronization complete'
# ----------------------------------------------------------------------
# Start application
# ----------------------------------------------------------------------
print 'Starting application', app_name
app_mgr  = AdminControl.queryNames('type=ApplicationManager,process='\
   + app_srv + ',*')
AdminControl.invoke(app_mgr,'startApplication',app_name)
print 'Application', app_name, 'started'
```

See notes

See notes

Notes:

1. Where these are the same basic variables as used before:
   - *aaaa* -- the application server long name
   - *bbbb* -- the node long name
   - *cccc* -- the location and name where you stored the `MyIVT.ear` file.
   - *dddd* -- is the application name: `My_IVT_Application` for the `MyIVT.ear` file.
2. The values for the `-BindJndiForEJBNonMessageBinding` option. This is what applies the JNDI name to the session bean in the MyIVT application. So that you may correlate the variables names to the values in sample `MyIVT.ear` file, we left the actual values in the sample above. The one difference is the `ejb_jndi` variable, which we're setting to something different from what's default in the packaged EAR.

   Note: For the values you'd use from *your* EAR file you need to run the command:

   ```
   print AdminApp.taskInfo('aaaa','BindJndiForEJBNonMessageBinding')
   ```

   command to get the values you'd use to install your application. In this case *aaaa* is the location and name of your EAR file.
3. The values for the `-MapEJBRefToEJB` option. The MyIVT application has one web module which references the one EJB. This is what tells the web module what JNDI name to look up to invoke the EJB. The values here what's found by default in the `MyIVT.ear` file. For your application you would need to modify them to match the values found in your application. Use the `AdminApp.taskInfo()` command and specify `MapEJBRefToEJB` as the option name.

   Note how in this example we are using the JNDI name applied to the EJB to be the reference made from the web module. We do that by assigning the value of `ejb_jndi_ref` equal to the value of `ejb_jndi`.
4. The first interior option list variable is built.
5. The second interior option list variable is built.
6. The overall options variable is then built, which incorporates the two interior list variables and the other options.
7. The application is installed, the nodes synchronized and the application started. This is the same as has been shown before.

☐ FTP that file to the z/OS file system in binary node and make sure the permissions allow WSADMIN to access it.

☐ Invoke that Jython file with the following command:

```
execfile('/aaaa/app9.jy')
```

Where *aaaa* is the location of your script file. You should see output similar to the earlier installation and uninstallation scripts.

Review	This exercise looked more complex because we were dealing with two options rather than one. So we had to build to "interior" option list variables. It showed how that is done, and it showed how the JNDI name for an EJB can be set and then have a reference to that JNDI be done by another module.

Exercise: map application to different virtual host at installation time

Objective	To show how a web module in an application can be bound to a different Virtual Host. This is largely the same type of exercise as for the other options This exercise assumes the new virtual host is already created ... this script does *not* create the new VH.

☐ Uninstall the MyIVT application from the cell. You may do this with the Admin Console, or the WSADMIN uninstall script you created earlier. Be sure that you save the changes and synchronize.

Note: If you uninstall using the Admin Console, you may need to exit and reestablish your WSADMIN prompt. There's some caching that goes on. A way to work around this is to issue the `AdminConfig.save()` command from the WSADMIN prompt. That may flush the cache and allow you to proceed without reestablishing the WSADMIN prompt.

☐ Create a file on your workstation called `app10.jy` and add the following to it:

Note: The file `app10.jy` (and all exercise scripts that are shown in this document) is supplied as part of the ZIP file that accompanies the techdoc on `ibm.com/support/techdocs`.

```
    # ---------------------------------------------------------------------
    # Set basic variables
    # ---------------------------------------------------------------------
    app_srv     =  'aaaa'
    app_node    =  'bbbb'
    app_ear     =  'cccc'   1
    app_name    =  'dddd'
    # ---------------------------------------------------------------------
    # Set variables for -MapWebModToVH
    # ---------------------------------------------------------------------
    web_mod       =  'My_IVT_Webapp_Display_Name'
    web_uri       =  'MyIVTWebApp.war,WEB-INF/web.xml'
2   web_vh        =  'proxy_host'
    print 'Value of -MapWebModToVH option list:',options
    # ---------------------------------------------------------------------
    # Build -MapWebModToVH option list
    # ---------------------------------------------------------------------
    web_vh_opt  =  '[[ ' + web_mod + ' ' + web_uri +\
3     ' ' + web_vh + ' ]]'
    print 'Value of web_vh_opt:',web_vh_opt
    # ---------------------------------------------------------------------
    # Build AdminApp.install() option list
    # ---------------------------------------------------------------------
    options = '[-node ' + app_node +\
4    ' -server ' + app_srv +\
     ' -MapWebModToVH ' + web_vh_opt + ']'
    print 'Value of overall option list:',options
    # ---------------------------------------------------------------------
    # Install application
    # ---------------------------------------------------------------------
    print 'Installing application', app_name
5   AdminApp.install(app_ear,options)
    AdminConfig.save()
    print AdminApp.list()
```

Notes:

1. Where these are the same basic variables as used before:
   - *aaaa* -- the application server long name
   - *bbbb* -- the node long name
   - *cccc* -- the location and name where you stored the `MyIVT.ear` file.
   - *dddd* -- is the application name after the application is installed.
2. The values for the `-MapWebModToVH` option. What you see are the attributes for that option as reported by `AdminApp.taskInfo()` for option `MapWebMobToVH` for the `MyIVT.ear` file. The value assigned to the variable `web_vh` is the *new* virtual host we wish to assign this application's web module to.

Note: This script assumes the VH named by that string already exists.

3. The list variable for the `-MapWebModToVH` option is built
4. The overall `AdminApp.install()` option list variable is built.
5. The application is installed and saved, as we've seen many times before.

Note: This script does *not* synchronize the nodes or start the application. You could easily do that as shown in the `app9.jy` file. We shortened this script to keep things a little easier.

☐ FTP that file to the z/OS file system in binary node and make sure the permissions allow WSADMIN to access it.

☐ Invoke that Jython file with the following command:

```
execfile('/aaaa/app10.jy')
```

Where *aaaa* is the location of your script file. You should see output similar to the earlier scripts.

Review Mapping an application's web module to a different virtual host is very similar to any of the other options. It's a matter of constructing the "interior" option list variable, then building the outer option variable and invoking the installation.

Example: map resource reference to connection factory JNDI

Objective To illustrate how a resource reference in an application is mapped to the JNDI name of a connection factory (CF) for a CICS, IMS or other non-relational connector.

Note: This is an *example*, not an exercise. Providing a sample application that will work with *your* CICS system is difficult. SuperSnoop and MyIVT had no data resource connections so they were assured to work. Plus, by this point you should understand the basics of coding more options on the `AdminApp.install()` function. This is more of the same, though it does use a different task name.

- We ran a "dummy" installation of an application in the Admin Console and got from the command assistance function the following for the resource reference binding to the connection factory:

```
-MapResRefToEJB [[ BeCashAcEJB BeCashAcSession
BeCashAcEJB.jar,META-INF/ejb-jar.xml CICSConnectionFactory
javax.resource.cci.ConnectionFactory CICSConnectionFactory "" "" ]]
```

- Not sure of what all those things referred to, we ran `AdminApp.taskInfo()` against the EAR file and supplied the `MapResRefToEJB` task name. We received the following:

```
The current contents of the task after running default bindings are:
Module: BeCashAcEJB
EJB: BeCashAcSession
URI: BeCashAcEJB.jar,META-INF/ejb-jar.xml
Resource Reference: CICSConnectionFactory
Resource type: javax.resource.cci.ConnectionFactory
Target Resource JNDI Name: CICSConnectionFactory
Login configuration name: null
Properties:
```

> **Note:** This particular application uses the string `CICSConnectionFactory` in a somewhat confusing way. We see it twice in the EAR file: for "Resource Reference" and for "Target Resource JNDI Name."
>
> The "Resource Reference" is the reference made in the application code itself. You can think of this like a `DDNAME` -- a symbolic reference that is to be resolved outside the source code. This application used the string `CICSConnectionFactory` as its "DDNAME" reference. This is the string found in the `<resource-ref>` tag in the deployment descriptor.
>
> The application also resolved that reference to a connection factory JNDI name of the same value: `CICSConnectionFactory`, which is what "Target Resource JNDI Name" is referring to. The developer assumed the real CF would be given a JNDI name of that string. This is the value we'll change to match our actual CF JNDI value.

- From that we determined the syntax of the option is:

```
-MapResRefToEJB [[
   BeCashAcEJB                               ◄···· Module
   BeCashAcSession                           ◄···· EJB
   BeCashAcEJB.jar,META-INF/ejb-jar.xml      ◄···· URI
   CICSConnectionFactory                     ◄···· Resource Reference
   javax.resource.cci.ConnectionFactory      ◄···· Resource type
   CICSConnectionFactory                     ◄···· Target Resource JNDI Name
   ""                                        ◄···· Login configuration name
   ""                                        ◄···· Properties
]]
```

With "Target Resource JNDI Name" being the JNDI name of the Connection Factory to which this resource reference is to be bound.

- Armed with this knowledge, we can build some Jython to construct this option:

```
# -------------------------------------------------------------------
# Set -MapResRefToEJB variables
# -------------------------------------------------------------------
ref_mod     =  'BeCashAcEJB'
ref_ejb     =  'BeCashAcSession'
ref_uri     =  'BeCashAcEJB.jar,META-INF/ejb-jar.xml'
res_ref     =  'CICSConnectionFactory'
res_type    =  'javax.resource.cci.ConnectionFactory'
CF_jndi     =  'eis/CICS_Region_A'    ⇦ the CF's JNDI name
log_cfg     =  '""'
ref_props   =  '""'
# -------------------------------------------------------------------
# Build -MapResRefToEJB context root option list
# -------------------------------------------------------------------
cf_ref_opt    =  '[[ ' + ref_mod + ' ' + ref_ejb +\
  ' ' + ref_uri + ' ' + res_ref + ' ' + res_type +\
  ' ' + CF_jndi + ' ' + log_cfg + ' ' + ref_props +\
  ' ]]'
print 'Value of -MapResRefToEJB interior option list:',cf_ref_opt
# -------------------------------------------------------------------
# Build AdminApp.install() option list
# -------------------------------------------------------------------
options = '[-node ' + app_node +\
 ' -server ' + app_srv +\
 ' -MapResRefToEJB ' + cf_ref_opt + ']'
```

Note:
- The structure of that is pretty much the same as we used for the other options, such as the context root of the web module, or the JNDI name given to an EJB module. The variable names were modified.
- Some may wonder if an EAR file could be programmatically queried for most of that information so that the script could automatically fill in the specific information such as the module and URI. The answer is yes -- you could use `AdminApp.taskInfo()` and place the output into a list variable, then loop into the list and find the information. That's just more advanced Jython, but not more WSADMIN. Here we've discovered the syntax of the `–MapResRefToEJB` option, which was our objective.

Review	We showed how the command assistance function of the Admin Console once again proved useful in determining the basic syntax of the option. `AdminApp.taskInfo()` against the EAR file provided the values for the key attributes. From there it was simply a matter of coding Jython to construct the `AdminApp.install()` command.

Example: map resource reference to JDBC data source

Objective	To illustrate how a resource reference in an application is mapped to the JNDI name of a JDBC data source.

There's no difference between a resource reference to a connection factory JNDI name and a reference to a JDBC data source JNDI name. Both use the `–MapResRefToEJB` option within the `AdminApp.install()` command. The process to determine the values in the EAR are the same as used for the CICS example just shown.

Review	A JDBC data source is like a JCA connection factory in terms of JNDI lookup and WSADMIN. If you understand one, you have the essence of the other.

Lesson wrap-up and summary

We just ran through a rather length set of exercises to see how to modify different settings for an application at installation time. They all involved essentially the same things:

- Using `AdminApp.options()` to query for the options applicable to a given EAR file.
- Using `AdminApp.taskInfo()` to query the contents of the EAR file so we could know how to properly point to the artifact in the EAR to modify
- Using the "command assistance" feature of WAS V6.1's Admin Console to see what the particular option's syntax looks like. That might not produce exactly what you need, but it'll be close. With that you can modify the syntax to fit your needs.
- Constructing a list variable to contain that option's attributes
- Constructing the overall options list variable
- Invoking installation using `AdminApp.install()`

We saw how there is an opportunity to modularize the scripts. The node synchronization routine we wrote, which is generic and will synchronize all nodes in a cell programmatically, can be called with the `execfile()` routine. That saves embedding the same Jython code in each application installation script file.

Topic: WSADMIN and configuration-oriented activities

Lesson Overview

This set of exercises revolves around the `AdminConfig` object. It relates to the configuration of the WebSphere cell. With this you can do things like add a server, change the values for a server. We'll also explore the `AdminTask` object, which is new with V6.1. The `AdminTask` object makes doing certain things much easier than might otherwise be the case if done with just the `AdminConfig` object.

Exercise: listing configuration "types"

Objective	To see how to query WebSphere to report back regarding things within its configuration. There are many configuration elements in a cell, not just nodes and servers. To change or add something you first tell WSADMIN what part of the configuration to focus in on.

☐ At the WSADMIN command prompt, issue the following command:

```
print AdminConfig.types()
```

You should get back a *very long* list:

```
AccessPointGroup
ActivationSpec
ActivationSpecTemplateProps
ActivitySessionService
AdminObject
   :
WorkManagerInfo
WorkManagerProvider
WorkManagerService
WorkloadManagementServer
```

Note: "Types" are configuration elements within the cell. As of this writing, there are 569 of them. Changes to the configuration are made against types. If you scroll through that list you'll see the more recognizable ones -- `Server`, `Node`, `ClusterMember`.

☐ Issue the following command:

```
print AdminConfig.list('Server')
```

For our test cell, we received back:

```
dmgr(cells/fzcell/nodes/fzdmnode/servers/dmgr|server.xml#Server_1)
fzsr01c(cells/fzcell/nodes/fznodec/servers/fzsr01c|server.xml#Server_1165007710549)
fzsr02c(cells/fzcell/nodes/fznodec/servers/fzsr02c|server.xml#Server_1165802936177)
nodeagent(cells/fzcell/nodes/fznodec/servers/nodeagent|server.xml#Server_1165008639793)
```

Note: Two things are worth noting: one, this is *all* the servers in the cell, including the DMGR and node agents; and two, the format of that name is longer than we normally use to refer to servers. We'll use the `AdminTask` object to list just application servers; and the `AdminConfig.showAttribute()` method can be used to return the more familiar long name.

☐ Issue the following command:

```
print AdminConfig.list('Node')
```

You should get back a similar list, but for the nodes in your cell.

☐ Create a file called `config1.jy` and supply in it the following:

Note: This file supplied in the ZIP that accompanies the Techdoc.

```
list_servers = AdminConfig.list('Server').split("\n")
for i in list_servers:
 long_name = AdminConfig.showAttribute(i,'name')
 print 'The more familiar long name is',long_name
print 'Done!'
```

Note: The `.split("\n")` suffix takes the output of `AdminConfig.list()`, which is a long string, and splits it into elements in list variable. Then a simple loop processes through the elements of the list variable. The `AdminConfig.showAttribute()` function returns the more common "long name" of a configuration object, given its longer unique name.

☐ Save the file and FTP it to your system. Make sure the permissions allow WSADMIN to read the file. Then issue the command:

```
execfile('/aaaa/config1.jy')
```

Where *aaaa* is the location of your script file. You should see something like this:

```
The more familiar long name is dmgr
The more familiar long name is fzsr01c
The more familiar long name is fzsr02c
The more familiar long name is nodeagent
Done!
```

☐ That was *all* the servers. Let's now look at how to get a list of just application servers in a given node. Create a file called `config2.jy` and supply in it the following:

```
# ----------------------------------------------------------------------
# Set basic variables
# ----------------------------------------------------------------------
node        =  'aaaa'
server_type =  'APPLICATION_SERVER'
# ----------------------------------------------------------------------
# Build AdminTask.listServers() command option list
# ----------------------------------------------------------------------
opt_list = '[-serverType ' + server_type + ' -nodeName ' + node + ']'
# ----------------------------------------------------------------------
# Create list variable with appservers in specified node
# ----------------------------------------------------------------------
appsvr_list = AdminTask.listServers(opt_list).split("\n")
for i in appsvr_list:
 long_name = AdminConfig.showAttribute(i,'name')
 print 'Appserver long name is',long_name
print 'Done!'
```

where *aaaa* is the long name of the node for which you wish to list the application servers.

☐ Save the file and FTP it to your system. Make sure the permissions allow WSADMIN to read the file. Then issue the command:

```
execfile('/aaaa/config2.jy')
```

Where *aaaa* is the location of your script file. You should see a list of the application servers in your node. The format should be the familiar long name.

Review	A brief tour of the `AdminConfig.list()` and `AdminTask.listServers()` functions. The InfoCenter can provide additional examples.

Exercise: add an application server

Objective	To show how a new application server can be added to a cell using WSADMIN. This will make use of the new `AdminTask` object, which greatly simplifies the process over what was required in the past. We determined the basic structure of the syntax using the Admin Console and the "command assistance" feature.

☐ Create a file called `config3.jy` and supply in it the following:

Note: This file supplied in the ZIP that accompanies the Techdoc.

```
    # ---------------------------------------------------------------------
    # Set basic variables
    # ---------------------------------------------------------------------
    node          =     'aaaa'
    server_long   =     'bbbb'
1   server_short  =     'CCCC'
    cluster_tran  =     'DDDD'
    template      =     'defaultZOS'
    # ---------------------------------------------------------------------
    # Build AdminTask option list variable
    # ---------------------------------------------------------------------
    options = '[-name ' + server_long +\
              ' -templateName ' + template +\
2             ' -specificShortName ' + server_short +\
              ' -genericShortName ' + cluster_tran + ']'
    # ---------------------------------------------------------------------
    # Invoke AdminTask command
    # ---------------------------------------------------------------------
3   AdminTask.createApplicationServer(node,options)
    AdminConfig.save()
```

Notes:

1. These are the basic variables needed by the command:
   - *`aaaa`* -- the node long name where the server will be built
   - *`bbbb`* -- the new server long name
   - *`CCCC`* -- the new server short name. This is referred to as the "Server specific short name" in the Admin Console. This must be upper-case.
   - *`DDDD`* -- the new server's "cluster transition name." This is referred to as the "Server generic short name" in the Admin Console. Also upper-case.

   Note: There is only one default z/OS template, and that's `'defaultZOS'` as shown. Other templates can be created, and if they're present they could be named here. But first starting out, all you'll see if `defaultZOS`.

   We *could* have hard-coded that into the script that constructed the `options` variable. We decided to keep it as a basic variable.

2. We build the option list for the `AdminTask.createApplicationServer()` command. We're using the variables set at the top of the script to populate the option list, and we're using the backslash character to break the line as shown.

3. We then invoke the command, providing the node as one attribute to the command and the list variable we just created as the second attribute. We then issue the AdminConfig.save() command to save the changes.

Notes:
- This script does not show the changes being synchronized. We've already shown many examples of how a script can call another script which invokes synchronization for the cell.
- The port assignments will be default, which means they most likely won't map to your port allocation standards. We'll see how to change those in a bit.
- You *may* require additional RACF (or other SAF product) profiles under this server to support starting it. Specifically, RACF `STARTED`, `SERVER` and `CBIND` profiles. This document will not explore that. See WP100653 on `ibm.com/support/techdocs` for more on RACF profiles that support starting a server.

☐ Save the file and FTP it to your system. Make sure the permissions allow WSADMIN to read the file. Then issue the command:

```
execfile('/aaaa/config3.jy')
```

Where `aaaa` is the location of your script file. Go the Admin Console to verify that the server is now present.

Note: Again, it'll be saved but not synchronized. The ports are not yet remapped. And there may not be sufficient RACF support under the new server definition to allow it to run. But the Admin Console will still show the new server, which will verify the script performed its function.

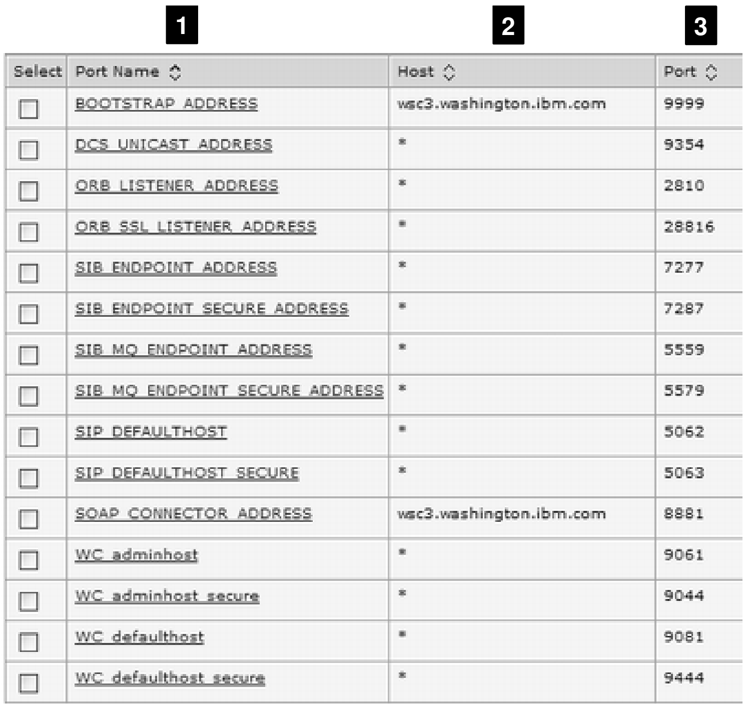
Review This exercise shows the power of the new `AdminTask` object. In past versions of the product it would have taken a great deal more effort to create the server. This also illustrated the value of the new "command assistance" feature of the Admin Console. We saw the structure of the `AdminTask` syntax by creating a dummy server, capturing the syntax, then discarding the new server before it was saved. From that we built the script.

Exercise: remap a server's TCP ports

Objective The creation of a server using WSADMIN will leave the server's ports at some default setting. Here we'll see how to programmatically modify the server's ports and set them to match your allocation method. Once again, we discovered the basic syntax of the `modifyServerPort` method by using the Admin Console's "command assistance" feature. We went into a server, changed a port value, captured the syntax, then discarded the change.

Note: What we're going to show you is a rather manual method of determining what all the `-endPointName` values are for a server. There are fancy programmatic ways to list those out and programmatically loop through. But for the sake of this document, we'll show the more intuitive way.

☐ In the Admin Console, navigate to your new server and click on the "Ports" link, which is on the right side of the screen, about a one page down. You should see *something* that looks like this:



Select	Port Name	Host	Port
☐	BOOTSTRAP_ADDRESS	wsc3.washington.ibm.com	9999
☐	DCS_UNICAST_ADDRESS	*	9354
☐	ORB_LISTENER_ADDRESS	*	2810
☐	ORB_SSL_LISTENER_ADDRESS	*	28816
☐	SIB_ENDPOINT_ADDRESS	*	7277
☐	SIB_ENDPOINT_SECURE_ADDRESS	*	7287
☐	SIB_MQ_ENDPOINT_ADDRESS	*	5559
☐	SIB_MQ_ENDPOINT_SECURE_ADDRESS	*	5579
☐	SIP_DEFAULTHOST	*	5062
☐	SIP_DEFAULTHOST_SECURE	*	5063
☐	SOAP_CONNECTOR_ADDRESS	wsc3.washington.ibm.com	8881
☐	WC_adminhost	*	9061
☐	WC_adminhost_secure	*	9044
☐	WC_defaulthost	*	9081
☐	WC_defaulthost_secure	*	9444

Notes:

1. The "Port Name" column contains the names we need for the `modifyServerPort` command's `-endPointName` option. What we'll do is capture them and code them into our WSADMIN script.
2. The "Host" column contains information we need for the `-host` option. Notice that most contain simply an asterisk, though some (`BOOTSTRAP` and `SOAP`) contain the specific host value. For our remapping exercise we'll maintain the values that got assigned by default.
3. The "Port" column shows what's currently assigned. We'll use the `-port` option to remap the value to our preferred value.

☐ Using the copy/paste facility of the browser, capture those "Port Name" values into a text file.

☐ Create a file called `config4.jy` and supply in it the following. Make sure the port name values you just captured are all accounted for in the script:

Note: This file supplied in the ZIP that accompanies the Techdoc.

```
# --------------------------------------------------------------------
# Set basic variable
# --------------------------------------------------------------------
svr  = 'aaaa'  1
# --------------------------------------------------------------------
# Create and invoke AdminTask command
# --------------------------------------------------------------------
AdminTask.modifyServerPort(svr,\
 '[-endPointName BOOTSTRAP_ADDRESS -port xxxx -modifyShared true]')  2
AdminTask.modifyServerPort(svr,\
 '[-endPointName SOAP_CONNECTOR_ADDRESS -port xxxx -modifyShared true]')
AdminTask.modifyServerPort(svr,\
 '[-endPointName WC_adminhost -port xxxx -modifyShared true]')
AdminTask.modifyServerPort(svr,\
 '[-endPointName WC_defaulthost -port xxxx -modifyShared true]')
AdminTask.modifyServerPort(svr,\
 '[-endPointName DCS_UNICAST_ADDRESS -port xxxx -modifyShared true]')
AdminTask.modifyServerPort(svr,\
 '[-endPointName ORB_SSL_LISTENER_ADDRESS -port xxxx -modifyShared true]')
AdminTask.modifyServerPort(svr,\
 '[-endPointName WC_adminhost_secure -port xxxx -modifyShared true]')
AdminTask.modifyServerPort(svr,\
 '[-endPointName WC_defaulthost_secure -port xxxx -modifyShared true]')
AdminTask.modifyServerPort(svr,\
 '[-endPointName SIP_DEFAULTHOST -port xxxx -modifyShared true]')
AdminTask.modifyServerPort(svr,\
 '[-endPointName SIP_DEFAULTHOST_SECURE -port xxxx -modifyShared true]')
AdminTask.modifyServerPort(svr,\
 '[-endPointName SIB_ENDPOINT_ADDRESS -port xxxx -modifyShared true]')
AdminTask.modifyServerPort(svr,\
 '[-endPointName SIB_ENDPOINT_SECURE_ADDRESS -port xxxx -modifyShared true]')
AdminTask.modifyServerPort(svr,\
 '[-endPointName SIB_MQ_ENDPOINT_ADDRESS -port xxxx -modifyShared true]')
AdminTask.modifyServerPort(svr,\
 '[-endPointName SIB_MQ_ENDPOINT_SECURE_ADDRESS -port xxxx -modifyShared true]')
AdminTask.modifyServerPort(svr,\
 '[-endPointName ORB_LISTENER_ADDRESS -port xxxx -modifyShared true]')
# --------------------------------------------------------------------
AdminConfig.save()
        3
```

Notes:

1. For our purposes only one basic variable is needed -- the server long name.
2. Rather than try to come up with a complicated looping structure (which would be possible, but more challenging), we opted to simply invoke the `AdminTask` command as many times as the server had ports.

   Change the `xxxx` in each line to the port value you wish for that endpoint.

   The `-modifyShared true` parameter is there because some of the endpoints are considered "shared" by WebSphere and can't be remapped unless that switch is present. Not all endpoints need it, but having it on all lines is insurance.

   It's not strictly necessary to break the line as we show here. We show that so the type size in this document doesn't get too small.
3. The changes are saved.

☐ Save the file and FTP it to your system. Make sure the permissions allow WSADMIN to read the file. Then issue the command:

```
execfile('/aaaa/config4.jy')
```

Where *aaaa* is the location of your script file. Go the Admin Console to verify that the server's ports have been remapped.

Note: Saved but not synchronized. And if the server was up when that script was run, it'll need to be stopped and restarted to pick up the change.

Review Changing a server's port values is tedious but not really that difficult. A looping structure could have been built. But it would have required building list variables for the end point names and the associated port values. What we showed here will work for nearly all new servers created. You would need to modify it slightly for node agents, as they have different ports.

Background: change node host name and node system name reference

Note: This is something desired by people who are doing disaster recovery testing and they want to bring up their system image on a different Sysplex. But a WebSphere configuration will not start if the host names and system names in the configuration XML don't match the actual seen on the system. In the past this required some rather extensive manual modification of the configuration XML files, which was difficult and prone to error. It has been made much easier with AdminTask.

☐ See WP100792 at `ibm.com/support/techdocs`. It provides a complete explanation of the process involved.

Exercise: create a new cluster across two nodes

Objective To show how WSADMIN could be used to create a cluster consisting of two cluster members across two nodes in a cell. It turns out this process is really a three step process:

1. Create the cluster using the `AdminTask.createCluster()` function
2. Add the first member using the `AdminTask.createClusterMember()` function
3. Add the second member using `AdminTask.createClusterMember()` function again, but with slightly different values

For this example we're showing the creation of a cluster where the initial server is brand new. Another way to do it is to select and existing server to act as the initial server then clone that to form the cluster. We're not going to show that, but the Jython would be very similar. Use the "command assistance" feature if you want to see what the syntax of that would look like.

Following that, you can use the `modifyServerPort` method we just looked at to remap the default port assignments that are given to each of the members.

☐ First, take a look at what the "command assistance" feature returned when we built a dummy cluster in the Admin Console (example shown next). We reformatted this to make it more readable; in its original format it consisted of three long command lines. Try to get a handle on structure of the options and the list variables. Don't worry about what all this stuff actually does:

```
    AdminTask.createCluster('[-clusterConfig
     [-clusterName clustername -shortName FZSR04]]') 1


    AdminTask.createClusterMember('[-clusterName clustername
      -memberConfig [-memberNode fznodec -memberName fzsr04c
                     -memberWeight 2 -specificShortName FZSR04C]
      -firstMember [-templateName defaultZOS       2
                    -nodeGroup DefaultNodeGroup
                    -coreGroup DefaultCoreGroup]]')


    AdminTask.createClusterMember('[-clusterName clustername
      -memberConfig [-memberNode fznoded -memberName fzsr04d    3
                     -memberWeight 2 -specificShortName FZSR04D]]')
```

Notes:

1. The first command is `AdminTask.createCluster()` and it has within its parenthesis a set of nested options: `[-clusterConfig [ options ]]`. When we get to the Jython you'll see we build the inner list first, then the outer one. This simply creates an empty cluster with no server members.
2. The second command is `AdminTaskcreateClusterMember()`, and this is used to create the *initial* cluster member. This is comprised of two inner option lists nested inside of an outer list:

   `[-clusterName xxx -memberConfig[ options ] -firstMember [ options ]]`

   We'll build each inner option list separately, then construct the overall outer option list after that.
3. The first command is also `AdminTaskcreateClusterMember()`, but it only has one inner option list. That's because it does *not* perform the `-firstMember` operation, but rather does just the `-memberConfig` operation:

   `[-clusterName xxx -memberConfig[ options ]]`

   We'll build the inner option list just like in #2, and then wrapper it in the outer option list.

Please Read This is the nature of WSADMIN scripting. Objects and methods and options, with some options having sub-options. Thankfully we have the "command assistance" function, otherwise we'd have to hunt around for this syntax on our own. With "command assistance" we can get a sample of the syntax and create our more orderly Jython from that.

☐ Create a file called `config5.jy` and supply in it the following. Make sure the port name values you just captured are all accounted for in the script:

Note: This file supplied in the ZIP that accompanies the Techdoc.

```
# -----------------------------------------------------------------------
# Set basic variables for cluster itself
# -----------------------------------------------------------------------
cluster_long     =  'aaaa'
cluster_short    =  'BBBB'
first_node       =  'cccc'
second_node      =  'dddd'  1
first_mem_long   =  'eeee'
first_mem_short  =  'FFFF'
second_mem_long  =  'gggg'
second_mem_short =  'HHHH'
# -----------------------------------------------------------------------
# Set other default variables if necessary
# -----------------------------------------------------------------------
template         =  'defaultZOS'
node_group       =  'DefaultNodeGroup'
core_group       =  'DefaultCoreGroup'  2
first_mem_wgt    =  '2'
second_mem_wgt   =  '2'
# -----------------------------------------------------------------------
# Build createCluster option lists (one nested inside other)
# -----------------------------------------------------------------------
cc_inner_opt     = '[-clusterName ' + cluster_long +\
                   ' -shortName ' + cluster_short + ']'      3
cc_outer_opt     = '[-clusterConfig ' + cc_inner_opt + ']'
# -----------------------------------------------------------------------
# Build initial createClusterMember option lists -- two inner lists
#   nested inside a single outer list: [ [inner] [inner] ]
# -----------------------------------------------------------------------
cm1_inner_opt1   = '[-memberNode ' + first_node +\
                   ' -memberName ' + first_mem_long +\    4
                   ' -memberWeight ' + first_mem_wgt +\
                   ' -specificShortName ' + first_mem_short + ']'
cm1_inner_opt2   = '[-templateName ' + template +\
                   ' -nodeGroup ' + node_group +\
                   ' -coreGroup ' + core_group + ']'
cm1_outer_opt    = '[-clusterName ' + cluster_long +\
                   ' -memberConfig ' + cm1_inner_opt1 +\
                   ' -firstMember ' + cm1_inner_opt2 + ']'
# -----------------------------------------------------------------------
# Build second createClusterMember option list -- one inner list
#   nested inside a single outer list: [ [inner] ]
# -----------------------------------------------------------------------
cm2_inner_opt1   = '[-memberNode ' + second_node +\
                   ' -memberName ' + second_mem_long +\    5
                   ' -memberWeight ' + second_mem_wgt +\
                   ' -specificShortName ' + second_mem_short + ']'
cm2_outer_opt    = '[-clusterName ' + cluster_long +\
                   ' -memberConfig ' + cm2_inner_opt1 + ']'
# -----------------------------------------------------------------------
# Build commands and invoke to create cluster
# -----------------------------------------------------------------------
AdminTask.createCluster(cc_outer_opt)
AdminTask.createClusterMember(cm1_outer_opt)  6
AdminTask.createClusterMember(cm2_outer_opt)
AdminConfig.save()
```

Notes:

1. The basic variables for the cluster are set first. Recall that in this example we're building a cluster with two members across two nodes. The basic variables reflect that. The variable names are self-explanatory. Remember that short names must be in upper-case.
2. We include a set of variables that you probably won't need to change, but rather than hard-coding the values down in the script, we exposed them as variables and set them apart from the other variables you'll definitely change.

3. The first command is `AdminTask.createCluster()`. As stated earlier, this has a nested option list. Here we're building the option list variable for that command, though we don't actually invoke the command until the end of the script.

   Note: If this is confusing, go back and take a look at the actual syntax offered by the command assistance function. Mapping the Jython to the actual will help make sense of what's going on in the script.

4. We create the nested option structure for the `createClusterMember()` method for the initial server member. This is more complex than the second cluster member because it has an additional inner option list.
5. We create the second `createClusterMember()` option list.
6. We construct the actual `AdminTask` commands and invoke. When done, we save.

☐ Save the file and FTP it to your system. Make sure the permissions allow WSADMIN to read the file. Then issue the command:

```
execfile('/aaaa/config5.jy')
```

Where *aaaa* is the location of your script file. Go the Admin Console to verify that the cluster has been created.

Note: But no synchronization, and no remapping of the default port values. We've shown how to do that elsewhere in this document.

Review The value of creating a script to create a cluster can be debated. Doing that in the Admin Console might be quicker. However, if you have several cells you're trying to maintain as mirrors of one another, creating scripts to add servers and clusters may be the way to insure consistency.

The script we build was long and somewhat complicated. Again, that's the nature of WSADMIN scripting. The command assistance function was invaluable in figuring out the syntax of the command. The construction of the Jython was really just carefully taking the output from command assistance, creating variables, and then substituting the variables into the actual commands where our actual values were found.

Lesson wrap-up and summary

In this section we spent some time with the `AdminConfig` and `AdminTask` objects. They are the primary tools to create and modify configuration elements. In WebSphere, those configuration elements are called "types," and there are over 500 different types available. You can create or modify pretty much anything with WSADMIN. The only question is how much time you want to spend investigating the syntax and creating/debugging the script.

But the command assistance feature of the Admin Console is valuable for this purpose. We showed how we used it to get a feel for the basic syntax structure. Then we modified that so we could use variable substitution and make the script more flexible.

AdminTask is new with V6 and is *wonderful*. They have taken things that required more involved `AdminConfig` scripts and condensed them into powerful `AdminTask` functions. We barely scratched the surface of the `AdminTask` object.

Topic: WSADMIN and operation-oriented activities

Lesson Overview

In this section we'll explore how to control the operations of the WebSphere cell using WSADMIN. This involves using the `AdminControl` object. With this you can do things like start and stop servers.

Exercise: check the current state of a server

Objective	Let's say you want to programmatically check the status of a server process before you try to do some other action ... for example, starting an application or issuing a `START` for the server itself. You can use the `AdminControl.completeObjectName()` function to see if the process is there or not. If the results are null, then the process isn't up. If the results are greater than 0 bytes in length, then the process is present.

☐ At the WSADMIN command prompt, issue the following command:

```
comp_id = AdminControl.completeObjectName('type=Server,process=aaaa,*')
```

where *`aaaa`* is the long name of a server you know is up and running. You should get nothing in return ... just the WSADMIN prompt.

Note: To check the status of a Node Agent involves something slightly different. We'll see that in the `oper1.jy` exercise coming next.

☐ Now issue:

```
   print comp_id
```

You should get back the "complete ID" of the named server, which will look *something* like this:

```
WebSphere:name=fzsr01c,process=fzsr01c,platform=proxy,node=fznodec,j2eeType=J2EE
Server,version=6.1.0.4,type=Server,mbeanIdentifier=cells/fzcell/nodes/fznodec/se
rvers/fzsr01c/server.xml#Server_1165007710549,cell=fzcell,spec=1.0,processType=M
anagedProcess
```

In other words, a big long string ... clearly longer than 0 bytes.

☐ Issue:

```
   print len(comp_server)
```

That will produce the length of the variable, which will be somewhere in the neighborhood of 253 bytes long.

Note: If the server was down the length of that variable would be `0`.

☐ Let's create a script that will check the status of any server or node agent. The format for a server is slightly different than a node agent, so the script will need to be a bit more creative. You can strip this back and make it simpler if you wish.

Create a file called `oper1.jy` and supply in it the following.

Note: This file supplied in the ZIP that accompanies the Techdoc.

```
# ---------------------------------------------------------------------
# Set basic variables and choose 'type' ... 1=Server, 2=NodeAgent
# ---------------------------------------------------------------------
node_name    =  'aaaa'      1
process_name =  'bbbb'
process_type = 1
# ---------------------------------------------------------------------
# Create command and issue
# ---------------------------------------------------------------------
if process_type == 1:
 type = 'Server'
elif process_type == 2:       2
 type = 'NodeAgent'
 process_name = 'nodeagent'
# ---------------------------------------------------------------------
complete_id = AdminControl.completeObjectName('type=' + type +\
  ',node=' + node_name +\
  ',process=' + process_name +\     3
  ',*')
# ---------------------------------------------------------------------
length_id = len(complete_id)
if length_id == 0:
  print 'Process',process_name,'is not started'
elif length_id != 0:                              4
  print 'Process',process_name,'started'
```

Notes:

1. Where *aaaa* is the node long name where the process resides and *bbbb* is the server long name if the process is a server.

   Note: If the process is a Node Agent, the value gets set down in the if-then-else block under #2. Node Agents always have a fixed value for `nodeagent` for the process name. We could have had you just code that on the variable at the top.

   The variable `process_type` is set to either `1` or `2`, depending on whether it's a server process or Node Agent process.

   Note: The script is crude ... no error checking for "out of range" is done.

2. An *if-then-else* is performed. It takes one branch if a server process and another if a node agent process. The `Type` variable is set.
3. The variable `complete_id` is populated with the `AdminControl.completeObjectName()` function. This looks complex because there's a lot of variable substitution going on.
4. The length of the variable `complete_id` is checked. If `0` it means the process is not up. If `1` it means it is.

☐ Save the file and FTP it to your system. Make sure the permissions allow WSADMIN to read the file. Then issue the command:

`execfile('/`*`aaaa`*`/oper1.jy')`

Where *aaaa* is the location of your script file. It should report back the status of the process, either 'not started' or 'started.'

Note: This script could be made more intelligent. In particular, it could take parameters on invocation and process them with `sys.argv`. This might be one way to make the routine generic and pass in the server name, node name and process type.

Review This illustrated a way to determine if a process is started.

Exercise: start and stop a server

Objective	Starting and stopping servers involves using the `AdminControl.invoke()` method. To use this we need to specify the management bean (mBean) and issue the command we want -- `stop` or `launchProcess`. To stop a server you touch the mBean of the server itself; to start a server you must make contact with the mBean of the Node Agent, which will then issue the start command.

☐ Create a file called `oper2.jy` and supply in it the following:

```
# ----------------------------------------------------------------------
# Stop a server - set basic variables
# ----------------------------------------------------------------------
process_name     =  'aaaa'
node_name        =  'bbbb'     1
# ----------------------------------------------------------------------
# Get "complete ID" of the server process
# ----------------------------------------------------------------------
id_string = 'type=Server,node=' + node_name +\
            ',process=' + process_name + ',*'     2
complete_id = AdminControl.completeObjectName(id_string)
# ----------------------------------------------------------------------
# Stop the server
# ----------------------------------------------------------------------
print 'Stopping Server'
AdminControl.invoke(complete_id, 'stop')     3
```

Notes:

1. Where *`aaaa`* is the long name of the application server process you're trying to stop, and *`bbbb`* is the long name of the node in which the server exists.

   Note: The node value is not strictly needed for stopping a server, but we have it in there because it is needed to start a server. We'll have the two scripts -- stopping and starting a server -- be similar to one another.

2. The string used to uniquely identify the server process is constructed and the `AdminControl.completeObjectName()` function executed. This populates the variable `complete_id` with the long string that WebSphere uses to uniquely identify a server.
3. The server is stopped with the `AdminControl.invoke()` function, which takes as input the complete ID of the server to be stopped (we derived that in step #3) and the `'stop'` keyword string.

☐ Save the file and FTP it to your system. Make sure the permissions allow WSADMIN to read the file. Then issue the command:

`execfile('/`*`aaaa`*`/oper2.jy')`

Where *`aaaa`* is the location of your script file. Then go and check to make sure the server actually came down.

☐ Let's make a script to start a server. Create a file called `oper3.jy` and supply in it the following:

```
# ------------------------------------------------------------------
# Start a server - set basic variables
# ------------------------------------------------------------------
process_name     =  'fzsr01c'    1
node_name        =  'fznodec'
# ------------------------------------------------------------------
# Get "complete ID" of the Node Agent
# ------------------------------------------------------------------
id_string = 'type=NodeAgent,node=' + node_name +\
            ',process=nodeagent,*'                        2
complete_id = AdminControl.completeObjectName(id_string)
# ------------------------------------------------------------------
# Start the server
# ------------------------------------------------------------------
print 'Starting Server'
AdminControl.invoke(complete_id,'launchProcess','[' + process_name + ']')
                                                        3
```

Notes:

1. Where *aaaa* is the long name of the application server process you're trying to start, and *bbbb* is the long name of the node in which the server exists.
2. Here we're getting the "complete ID" of the Node Agent that supports server to be started. It is against the Node Agent that we issue the `AdminControl.invoke()` to start a server.

   Note: If the Node Agent is also down then you're out of luck. The Node Agent is one of those basic pieces of the infrastructure that must be up for this stuff to work. There's no way to start a Node Agent using WSADMIN.

3. The server is started with the `AdminControl.invoke()` function, which takes as input the complete ID of the Node Agent, the string `'launchProcess'` and the list variable that contains the server process name, which we set up in #1.

Review The script we created was pretty basic ... it did almost no error checking and it didn't check to see if the server was already up or down. But it is functional.

The AdminControl object has a `startServer` and a `stopServer` method. It would achieve the same effect. We went with the `AdminControl.invoke()` process because that's what the Admin Console's "command assistance" feature told us was used when we started and stopped the server from it.

Exercise: synchronize a node, multiple nodes, or whole cell

We've already done this. See "Exercise: node synchronization" on page 49. That script synchronized all the nodes in a cell by getting a list of all nodes then looping through it. You could modify the script to synchronize a specific node if you wished.

Lesson wrap-up and summary

We used the AdminControl function in several places throughout this document -- node synchronization, starting and stopping applications, etc. Here we showed that it can be used to start and stop servers themselves.

Further Exploration of the AdminTask Object

In this section we'll shift away from the "primer" format and simply explore the AdminTask object a bit more. The reason for this is because AdminTask supplies some remarkably powerful commands and it's useful to be aware of them.

The objectives of this section are to help you understand:

- How to use the general "help" facility to find out what AdminTask commands are available, and
- How to use the help facility to determine the synatx and usage of the command

Jython or JACL?

For this section we'll use Jython. If you try to run these commands yourself, don't forget to put `-lang jython` in the invocation string.

Listing all the different "command groups"

AdminTask "command groups" are simply logical groupings of different AdminTask. For example, here are *a few* of the command groups and the short description that is offered by the help facility:

ClusterConfigCommands - Commands for configuring application server clusters and cluster members.
ServerManagement - A group of command that configure servers
PortManagement - A group of admin commands that help in managing WebSphere ports

And more ... that's just three of about 65 ... but those are three we suspect will contain the most commonly used WSADMIN commands.

Where did that come from? By issuing the following command:

```
print AdminTask.help('-commandGroups')
```

What results is a long list of groups. By reading the command group names and their descriptions you can get a sense for where the command your looking for might reside.

Note: If you know the command you need, you can skip this command group thing. This is handy when you're not sure what command might apply.

Listing the commands within a command group

Let's say you know the group and you want to see the commands within the group. For instance, let's say the `ServerManagement` group caught your attention. Here's how you'd list all the commands in that group:

```
print AdminTask.help('ServerManagement')
```

That produces a list of commands:

```
 Commands:
changeClusterShortName - A command that can be used to change the cluster's short name.
changeServerGenericShortName - A command that can be used to change the server generic short name.
changeServerSpecificShortName - A command that can be used to change the server specific short name.
    :
   (middle of the list clipped to save space in this document)
    :
showServerInstance - Show Server Instance configuration. This command only applies to the z/OS platform.
showServerTypeInfo - Show server type information.
showTemplateInfo - A command that displays all the Metadata about a given template.
```

Note: The InfoCenter has some of these. It's worth a check to see if the InfoCenter lists the command you're interested in. If it's there, the information is has is excellent.

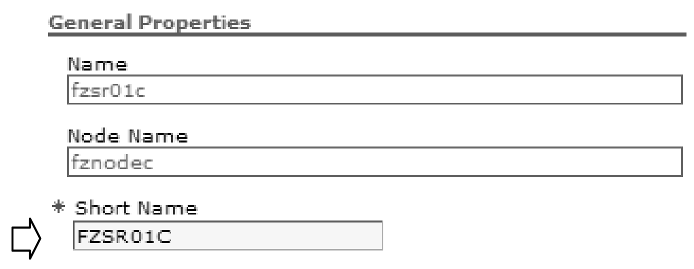
Listing the help on a specific command

Let's explore two of those:

- `changeServerSpecificShortName`

  The "specific short name" is a z/OS-only definition; it is the short name assigned to the application server. (The "generic short name" is something different ... the generic short name is sometimes known as the "cluster transition name," which is used by WebSphere as the WLM dynamic application environment name.)

  It looks like this in the Administrative Console, under the server's general properties:



General Properties

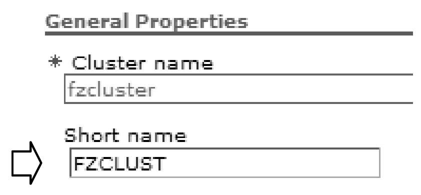
Name: fzsr01c

Node Name: fznodec

* Short Name: FZSR01C

- `changeClusterShortName`

  The short name for a cluster is used as the WLM application environment for each of the server members in the cluster. Here's what it looks like in the Administrative Console, under the *cluster's* general properties:



General Properties

* Cluster name: fzcluster

Short name: FZCLUST

changeServerSpecificShortName

Issuing `AdminTask.help('changeServerSpecificShortName')` yields:

```
Arguments:
  *serverName - The Server Name
  *nodeName - The Node Name
  *specificShortName - The server specific short name is applicable only on zOS
    platforms. This represents the specific short name of the server. All
    servers should have unique specific short name. This parameter is optional
    and when it is not specified a unique specific short name is automatically
    assigned. The value should be 8 chars or less and all upper case.
```

By itself that doesn't tell the syntax of the command. But based on other AdminTask commands we have seen we found the syntax is this:

```
AdminTask.changeServerSpecificShortName('[-serverName aaaaa
                                  -nodeName bbbbb -specificShortName ccccc]')
```

Where:
`aaaaa` - is the server long name you wish to work against
`bbbbb` - is the node long name in which the server resides
`ccccc` - is the server short name you wish to apply

It offers a null response -- two single quotes -- to indicate success.

Note: Don't forget the `AdminConfig.save()` and the synchronization if this is a network deployment configuration.

changeClusterShortName

Issuing `AdminTask.help('changeClusterShortName')` yields:

```
Arguments:
  *clusterName - The cluster short name is applicable only on zOS platforms.
    This represents the short name of the cluster. Every cluster should have
    unique short name. This parameter is optional and when it is not specified a
    unique short name is automatically assigned. The value should be 8 chars or
    less and all upper case.
  *shortName - The cluster short name is applicable only on zOS platforms. This
    represents the short name of the cluster. Every cluster should have unique
    short name. This parameter is optional and when it is not specified a unique
    short name is automatically assigned. The value should be 8 chars or less
    and all upper case.
```

Again, by itself that doesn't tell the syntax of the command. But just like the server short name command, we find the syntax here is similar:

```
AdminTask.changeClusterShortName('[-clusterName aaaaa -shortName ccccc]')
```

It offers a null response -- two single quotes -- to indicate success.

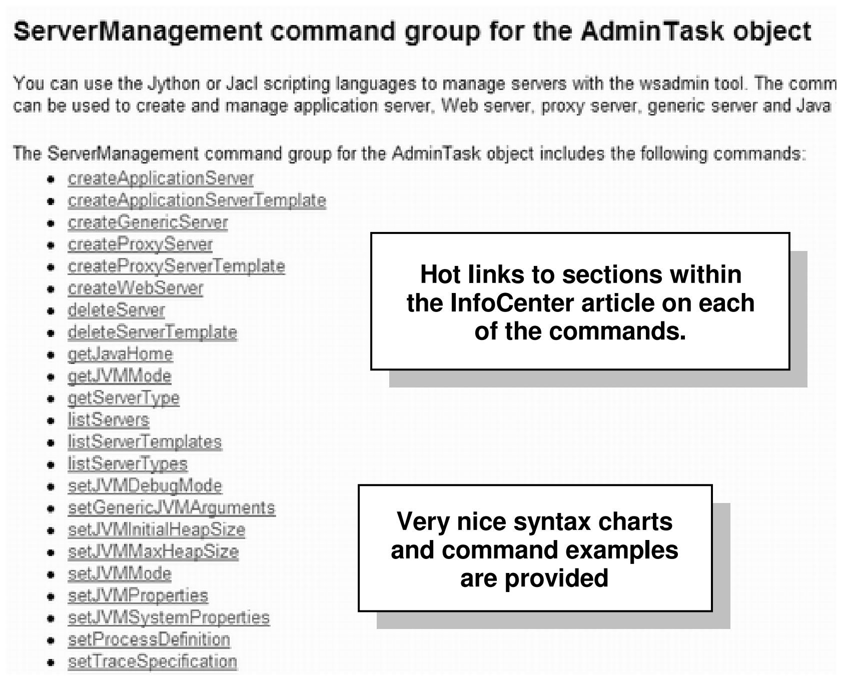
Note: Don't forget the `AdminConfig.save()` and the synchronization if this is a network deployment configuration.

What about the InfoCenter?

The inventory of AdminTask commands is being updated all the time, and sometimes additions don't make their way into the InfoCenter. When that happens you should open a PMR and specify that it is a defect in the documentation.

The two commands shown here, changeServerSpecificShortName and changeClusterShortName, were two that for whatever reason didn't make it into the InfoCenter. A document APAR is now open and that will get fixed ... in fact, when you read this it may well be fixed.

So, we would encourage you to look in the InfoCenter for command syntax because the InfoCenter is an excellent source of information on these commands. For example, a search on "ServerManagement" (one of the command group names) came up with this:



ServerManagement command group for the AdminTask object

You can use the Jython or Jacl scripting languages to manage servers with the wsadmin tool. The comm
can be used to create and manage application server, Web server, proxy server, generic server and Java

The ServerManagement command group for the AdminTask object includes the following commands:

- createApplicationServer
- createApplicationServerTemplate
- createGenericServer
- createProxyServer
- createProxyServerTemplate
- createWebServer
- deleteServer
- deleteServerTemplate
- getJavaHome
- getJVMMode
- getServerType
- listServers
- listServerTemplates
- listServerTypes
- setJVMDebugMode
- setGenericJVMArguments
- setJVMInitialHeapSize
- setJVMMaxHeapSize
- setJVMMode
- setJVMProperties
- setJVMSystemProperties
- setProcessDefinition
- setTraceSpecification

Hot links to sections within the InfoCenter article on each of the commands.
Very nice syntax charts and command examples are provided

Security Related Exercises

Lesson Overview

Security is a broad and complicated topic. We can't cover ever aspect of it. But we will touch on a few of the more common things people ask:

- May I use an ID other than the WebSphere Admin ID?
- May I invoke WSADMIN from a workstation and connect to WebSphere on z/OS?
- What is "fine grained security" and how do I use it?

Important! Do not make any RACF (or SAF) changes without first reviewing them with your security administrator. We strongly encourage you to be very careful with some of the exercises we have in this section.

Exercise: use a different ID from WebSphere Admin ID

Objective	To show what happens when you use an ID, other than the WebSphere Admin ID, when launching WSADMIN. There are potentially *three* levels of failure -- 1) file permission access failure; 2) SSL connection establishment failure; 3) SOAP or RMI port access failure. We'll show each one in turn. We're going to run through several exercises here: we'll show the first failure, then correct it. Then we'll show the second failure, and then correct it. Finally we'll show the last failure and correct it as well. When you're through you'll have authorized your new ID to use WSADMIN.

Error: File permission access failure

☐ Choose some ID, other than the WebSphere Admin ID, which you wish to use for this exercise. Perhaps your personal ID.

☐ Open up a telnet or OMVS prompt using the ID you chose.

☐ Change directories the following location:

```
/aaaa/DeploymentManager/profiles/default/bin
```

where *aaaa* is the mount point for the Deployment Manager node.

☐ Issue the **whoami** command to make sure you're operating as the ID you chose in the first step.

☐ Now issue the following command:

```
./wsadmin.sh -conntype NONE
```

Note: With `-conntype NONE` we remove from the equation the SSL connection and port access considerations. This reduces the issue down to just file access.

After a bit you should see a big long list of problems, with *something* like this included:

```
!MESSAGE Error reading configuration:
/<mount>/DeploymentManager/profiles/default/configuration/org
.eclipse.osgi/.manager/.fileTableLock (EDC5111I Permission denied.)
```
(plus a long Java stack trace which we're not showing here)

and:

```
WASX7448E: The trace file cannot be written to location
/<mount>/DeploymentManager/profiles/default/logs/wsadmin.traceout .
Please specify a different location with -tracefile option.
```

☐ Understand the reason why this occurred:

You attempted to launch WSADMIN from the Deployment Manager's configuration directory. That permissions on key files in that configuration directory typically have for "other" access *either `0` (no access) or `5` (read and execute, but not write). The `wsadmin.traceout` file in particular has permissions 775, which means "other" can't write to it.*

When we launched WSADMIN earlier under the authority of the WebSphere Admin ID, that ID was part of "group," not "other." Group access is `7`, which is "read, write and execute."

The way to overcome this problem is to connect the new ID to the WebSphere "Configuration Group" ID. We'll do that in an upcoming exercise.

Solution: File permission access failure

☐ Connect the ID to the WebSphere Configuration Group:

```
CONNECT aaaa GROUP(bbbb)
```

where:

- `aaaa` -- is the ID you're attempting to use; the one that is *not* the WebSphere Admin ID
- `bbbb` -- is the WebSphere Configuration Group. If you're not sure what that value is, check to see what groups the WebSphere Admin ID is connected to. You'll see the WebSphere Configuration Group listed. It'll probably have a name such as *`xx`*`CFG`, where *`xx`* is your two-character cell identifier.

Note: Connecting the ID to the WebSphere configuration group gives that ID authority your security administrator may not be comfortable with. Review this with them before you make the connection.

☐ Close out your telnet or OMVS session and reopen it.

Why? Group associations are often cached at initialization time. By closing and reopening the telnet or OMVS session you refresh the group association cache.

☐ Again, issue the following command from the telnet session:

```
./wsadmin.sh –conntype NONE
```

☐ You should see that proceed without trouble to the `wsadmin>` prompt.

☐ Issue the command **`quit`** to exit the WSADMIN session and return to the UNIX prompt.

☐ Understand why that fixed the problem:

By connecting the user to the WebSphere Configuration Group, you changed the access from "other" to "group". "Group" has the authority to write the necessary files to support WSADMIN; "other" did not.

Error: SSL connection establishment failure

☐ From the telnet or OMVS session, make sure you're still operating under the authority of the ID that is *not* the WebSphere Admin ID. Issue the **`whoami`** command to verify you are *not* the WebSphere Admin ID.

☐ Issue the following command (all on one line):

```
./wsadmin.sh –conntype SOAP –host aaaa –port bbbb
                                        –user cccc –password dddd
```

where:

- *`aaaa`* is the host address where the DMGR can be reached
- *`bbbb`* is the SOAP port of the DMGR
- *`cccc`* is the **WebSphere Admin ID**
- *`dddd`* is the password for the **WebSphere Admin ID**

Note: Yes, continue to use the WebSphere Admin ID for the `-user` and `-password` parameters. That's used to authorize access into the SOAP or RMI port. What we're testing *right now* is the keyring/certificate/SSL issue. That relates to the ID under which `wsadmin.sh` is operating.

You should *something* that looks like this:

```
*** SSL SIGNER EXCHANGE PROMPT ***
SSL signer from target host wsc3.washington.ibm.com is not found in
trust store safkeyring:///WASKeyring.FZCELL.

Here is the signer information (verify the digest value matches what
is displayed at the server):

Subject DN:    CN=wsc3.washington.ibm.com, OU=FZBASEC, O=IBM
Issuer DN:     CN=WAS CertAuth for Security Domain, OU=FZCELL
Serial number: 2
Expires:       Fri Dec 31 22:59:59 EST 2010
SHA-1 Digest:
A6:07:53:38:CC:5E:48:6A:11:F2:13:20:CF:21:80:D7:8C:71:0A:BC
MD5 Digest:    11:AB:A5:79:82:7A:66:33:5F:5B:E1:DB:95:19:A1:E4

Subject DN:    CN=WAS CertAuth for Security Domain, OU=FZCELL
Issuer DN:     CN=WAS CertAuth for Security Domain, OU=FZCELL
Serial number: 0
Expires:       Fri Dec 31 22:59:59 EST 2010
SHA-1 Digest:
A6:07:53:38:CC:5E:48:6A:11:F2:13:20:CF:21:80:D7:8C:71:0A:BC
MD5 Digest:    11:AB:A5:79:82:7A:66:33:5F:5B:E1:DB:95:19:A1:E4

Add signer to the trust store now? (y/n)
```

- ☐ Respond **`N`** to the prompt. WSADMIN does not have the ability to "add the signer to the trust store" when the trust store is SAF and the keyring has not yet been created.
- ☐ You should see *something* like the following:

```
WASX7213I: This scripting client is not connected to a server process;
please refer to the log file:
/<mount>/DeploymentManager/profiles/default/logs/wsadmin.traceout
for additional information.
WASX8011W: AdminTask object is not available.
WASX7031I: For help, enter: "print Help.help()"
wsadmin>
```

Note: It couldn't establish the SSL connection so it dropped back to a "conntype NONE" mode.

- ☐ At the `wsadmin>` prompt, issue the command **`quit`** to exit WSADMIN. You should be back at the UNIX prompt.
- ☐ Understand the reason why this occurred:

The establishment of an SSL connection requires that the client (WSADMIN, operating under the non-Admin ID you're using for this exercise) have a "trust store" with the Certificate Authority certificate that signed the DMGR's server certificate. The ID you're using may or may not have a keyring; it most likely does not have the CA certificate.

Solution: SSL connection establishment failure

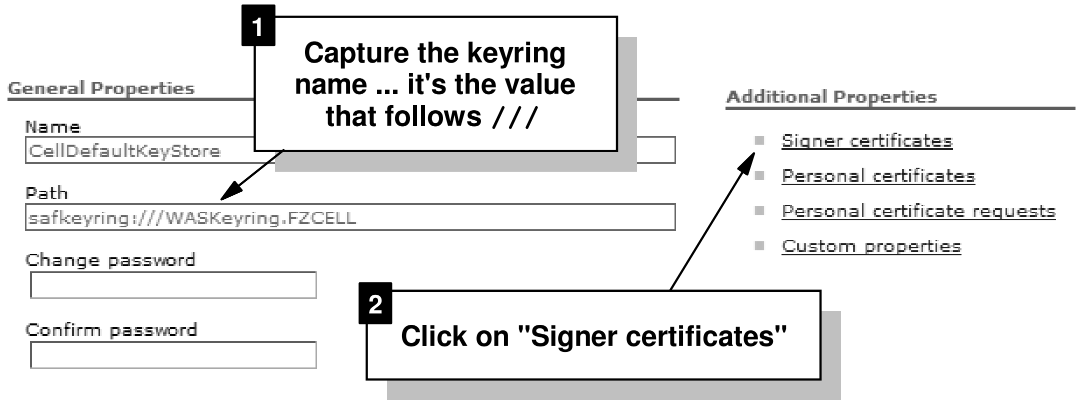
- ☐ We need to add a keyring to the ID and connect the CA certificate. If you know the values for those to things, then skip to the step that reads "Define a keyring for your non-Admin ID." on page 89. Otherwise, follow the instructions that follow here to capture that information:
  - ☐ Log onto the Admin Console

☐ Go to ***Security* ⇨ *SSL certificate and key management***.

☐ Look the right of the screen and click on ***Key stores and certificates***

☐ Click on the link ***CellDefaultKeyStore***

☐ Now do the following:


1
Capture the keyring name ... it's the value that follows ///
2
Click on "Signer certificates"
General Properties
Additional Properties
Signer certificates
Personal certificates
Personal certificate requests
Custom properties

Name: CellDefaultKeyStore

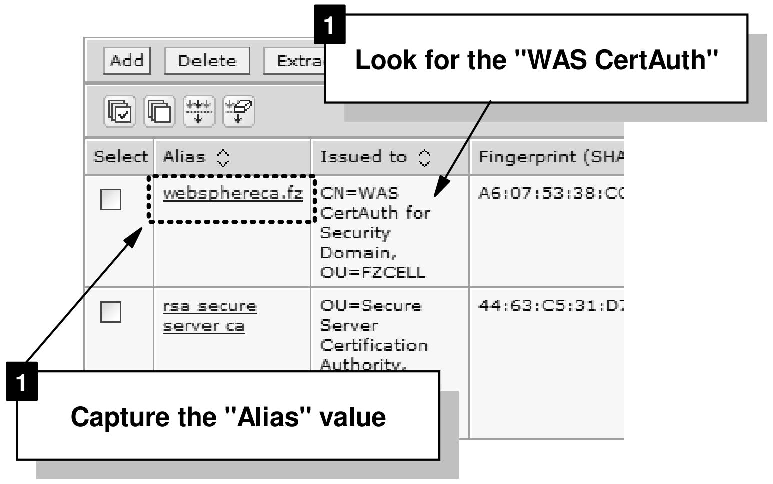
Path: safkeyring:///WASKeyring.FZCELL

Change password:

Confirm password:

Keyring Name *Value is case-sensitive*	

☐ You should see something like this under "Signer certificates". Do as indicated:



Select	Alias	Issued to	Fingerprint (SHA
☐	websphereca.fz	CN=WAS CertAuth for Security Domain, OU=FZCELL	A6:07:53:38:C
☐	rsa secure server ca	OU=Secure Server Certification Authority,	44:63:C5:31:D

CertAuth Alias Name	

Notes:
- If you are using a real-world Certificate Authority rather than the WebSphere CertAuth, then look for the real-world one you use.
- The "Alias" value will appear in all lower-case, but the value is in fact a mixed-case value. What we will do with this value requires that we know the exact case of the string. We're going to determine that next.

☐ Go to ISPF Option 6 and issue the following RACF command to list all the certificates associated with the WebSphere Admin ID. We'll do this to find the exact case for that CertAuth certificate:

RACDCERT ID(*aaaa*) LISTRING(bbbb)

where:

- *aaaa* -- is the WebSphere Admin ID
- *bbbb* -- is the Keyring name you captured a few steps back (case matters!)

You should see *something* like this:

```
Digital ring information for user FZADMIN
 Ring:
      >WASKeyring.FZCELL<
  Certificate Label Name             Cert Owner     USAGE      DEFAULT
  --------------------------------   ------------   --------   -------
  WebSphereCA.FZ                     CERTAUTH       CERTAUTH     NO
  Verisign Class 3 Primary CA        CERTAUTH       CERTAUTH     NO
  Verisign Class 1 Primary CA        CERTAUTH       CERTAUTH     NO
  RSA Secure Server CA               CERTAUTH       CERTAUTH     NO
  Thawte Server CA                   CERTAUTH       CERTAUTH     NO
  Thawte Premium Server CA           CERTAUTH       CERTAUTH     NO
  Thawte Personal Basic CA           CERTAUTH       CERTAUTH     NO
  Thawte Personal Freemail CA        CERTAUTH       CERTAUTH     NO
  Thawte Personal Premium CA         CERTAUTH       CERTAUTH     NO
  Verisign International Svr CA      CERTAUTH       CERTAUTH     NO
```

☐ Look for the "Certificate Label Name" that corresponds to the "Alias" you captured from the Admin Console. Capture the *exact case* here:

CertAuth Label Name: *In mixed case as seen*	

☐ Define a keyring for your non-Admin ID.. Use the following RACF command:

```
RACDCERT ADDRING(aaaa) ID( bbbb )
```

where:

- *aaaa* -- is the Keyring name for the cell. You captured this back on page 88.
- *bbbb* -- is the ID you're trying to enable to use WSADMIN

Note: Check with your security administrator prior to creating a keyring for the ID.

☐ Connect the "signer certificate" to the keyring. Use the following RACF command:

```
RACDCERT ID(aaaa) CONNECT (RING(bbbb) LABEL('cccc') CERTAUTH)
```

where:

- *aaaa* -- is the new ID you're trying to enable to use WSADMIN.
- *bbbb* -- is the Keyring name for the cell. You captured this back on page 88.
- *cccc* -- is the Certificate Authority label. You captured this on page 89.

Note: Check with your security administrator prior to connecting the cert to the keyring.

☐ Once again, issue the following command (all on one line):

```
./wsadmin.sh –conntype SOAP –host aaaa –port bbbb
                                         –user cccc –password dddd
```

where:

- *aaaa* is the host address where the DMGR can be reached
- *bbbb* is the SOAP port of the DMGR
- *cccc* is the **WebSphere Admin ID**
- *dddd* is the password for the **WebSphere Admin ID**

Note: Yes, continue to use the WebSphere Admin ID for the `-user` and `-password` parameters. That's used to authorize access into the SOAP or RMI port. What we're testing *right now* is the keyring/certificate/SSL issue. That relates to the ID under which `wsadmin.sh` is operating.

☐ If everything worked as designed, you should get:

```
WASX7209I: Connected to process "dmgr" on node <node> using SOAP
connector;  The type of process is: DeploymentManager
WASX7031I: For help, enter: "print Help.help()"
wsadmin>
```

☐ Issue the command **`quit`** to exit WSADMIN.

☐ Understand why what you did fixed the problem:

By giving this other ID a keyring with the same name as the keyring used by the DMGR, and by connecting to that ring the Certificate Authority label used to sign the DMGR's server certificate, you allowed the SSL connection to work. Now the DMGR's server cert, when presented to the WSADMIN client, can be verified. Before it could not because the new ID had no keyring or certificate.

Error: SOAP or RMI port access failure

☐ Once again, issue the following command (all on one line):

```
./wsadmin.sh -conntype SOAP -host aaaa -port bbbb
                                         -user cccc -password dddd
```

where:

- *`aaaa`* is the host address where the DMGR can be reached
- *`bbbb`* is the SOAP port of the DMGR
- *`cccc`* is the non-Admin ID
- *`dddd`* is the password for the non-Admin ID

☐ You should see the following:

```
WASX7209I: Connected to process "dmgr" on node <node> using SOAP
connector;  The type of process is: DeploymentManager
WASX7031I: For help, enter: "print Help.help()"
wsadmin>
```

No Error? That's right. That's because earlier you connected that ID to the WebSphere Config Group. The moment you did that, you inherited the authority of that Group. One of the authorities that Group ID had was access to something called `EJBROLE.administrator`. That's what controls administrative access to the management interface.

When the WSADMIN client is on MVS along with the DMGR, it's somewhat hard to create this SOAP or RMI access failure. We needed to connect the ID to the Group ID to overcome the most basic problem, which was file permission access. Once we did that, this problem went away.

If the WSADMIN client was on a remote workstation (which we discuss under "Exercise: WSADMIN client on a remote workstation" on page 93), then we could test the two things independently. How you grant an ID access to the EJBROLE profile is discussed next.

Background: how WebSphere controls administrative authority

When someone presents themselves to WebSphere and seeks entry to use the administrative services -- either through the Admin Console or WSADMIN -- they do so by presenting their userid and password. WebSphere then determines whether they will be granted access and how much authority they will have once they get in the front door.

Note: There's actually *two* levels of access here -- authority to access the port, and authority to access WebSphere administrative services. Access to the port is permitted (by default) based on a valid RACF id and password. So any valid RACF ID should be able to do that. But once inside the port, then access to WebSphere administrative services is based on what we're going to describe next -- `EJBROLE` access.

When a WebSphere cell is created, five `EJBROLE` profiles are created:

```
administrator
monitor
configurator
operator
deployer
adminsecuritymanager
```

Each one has different levels of authorization. For example, `monitor` defines only the ability to look at, but not change, the WebSphere configuration. `operator` is allowed to start and stop things, but not deploy new applications. `administrator` has nearly full authority.

What defines what authority an ID has to do WebSphere tasks is which `EJBROLE` the ID has been granted access to. Let's say you have a new ID of `SMITH`. You want `SMITH` to have the authority of `deployer`, but nothing else. So what you'd do is "permit" `SMITH` access to the EJBROLE:

```
PERMIT deployer CLASS(EJBROLE) ID(SMITH) ACCESS(READ)
```

Then when SMITH presents himself, WebSphere will check with RACF and see what `EJBROLE` `SMITH` has access to. It'll find that `SMITH` has access to `deployer`. That tells WebSphere what authority `SMITH` should be allowed.

When the WebSphere cell is first built, only two `PERMIT` operations are performed:

- The WebSphere Configuration Group ID is granted `READ` to `administrator`
- The WebSphere Admin ID is granted `READ` to `adminsecuritymanager`

It turns out that the Admin ID is also connected to the Configuration Group, which means the Admin ID inherits the authority of the group. The Admin ID therefore has access to *two* authorities: `adminsecuritymanager`, which it was granted explicitly; and `administrator` which it inherits from its connection to the Configuration Group ID.

In the earlier exercise you granted the non-Admin ID access to the Configuration Group ID. By doing that, you allowed that non-Admin ID to inherit the authority of the Group ID, which included access to `administrator`. Therefore, what was your "non-Admin ID" became, in effect, an Admin ID.

That's why using that ID on the `-user` and `-password` of the WSADMIN invocation worked. When that ID presented its credentials to WebSphere, a check was made against RACF which saw that the ID had access to the group, which meant it had access to `administrator`. That was enough to be allowed access to the WSADMIN interface.

Determining the EJBROLE definitions created for a cell

WebSphere for z/OS provides a way to qualify the EJBROLE profiles with a prefix. It's called a "Security Domain Identifier." For example, rather than merely `administrator`, it would be **FZ.**`administrator`, where `FZ` is the prefix used when the cell was created. This is a way to differentiate the EJBROLEs used for one cell from another.

Before you can grant a new ID permission to use a cell's EJROLEs, you need to know what domain identifier was used for a cell, if any was used at all. This can be done through the Admin Console:

Note: Do the following only if you want to find out the domain identifier for a cell. Do *that* only if you're interested in granting another ID access to the EJBROLE profiles for a cell.

☐ In the Admin Console go to:

Security ⇨ Secure administration, applications, and infrastructure

☐ Then click on the link ***Custom properties***, in the lower-right of the panel.

☐ At the bottom, click on the little arrow symbol to advance to page 2

☐ Near the bottom, locate the property `security.zOS.domainName`

☐ Note the value seen there, including the case:

Security Domain Identifier:	

Note: If the property doesn't appear, or appears with no value associated with it, then the cell does not use a "Security Domain Identifier." That means the EJROLEs will be just as listed before: `administrator`, `monitor`, `configurator`, etc.

Seeing what IDs have permission to an EJBROLE profile

☐ If you want to see what IDs have access to a given EJBROLE, you can issue the following command:

```
RLIST EJBROLE aa.bbbbbb AUTH
```

where:

- *`aa`* -- is the security domain identifier for the cell. The value is case sensitive.
- *`bbbbbb`* -- is the EJBROLE profile, for example `administrator`. Also case sensitive.

For example:

```
RLIST EJBROLE FZ.administrator AUTH
```

Or, if no security domain identifier:

```
RLIST EJBROLE administrator AUTH
```

Down at the bottom of the listing you'll see *something* like this:

```
USER       ACCESS    ACCESS COUNT
----       ------    ------ -----
SYSADM1    ALTER        000000
FZCFG      READ         000000
```

What that example is showing is that the `FZCFG` ID -- the configuration group ID for the cell - has `READ` access to the `FZ.administrator` role. And as we mentioned before, if you want to grant another ID access to the EJBROLE, and a group already has `READ` access to it, simply connect the ID to the group ... the ID will then inherit the authority of the group, including EJBROLE access.

Granting an ID permission to an EJBROLE profile

☐ You have two options for granting an ID access to an EJBROLE

Note: Check with your security administrator prior to making any changes to EJBROLE access.

- Grant the ID direct access with the following command:

  PERMIT *aa.bbbbbb* CLASS(EJBROLE) ID(*cccc*) ACCESS(READ)

  where:

  - *aa* -- is the security domain identifier for the cell. The value is case sensitive.
  - *bbbbbb* -- is the EJBROLE profile, for example `administrator`. Also case sensitive.
  - *cccc* -- is the ID you wish to grant access

- Grant some group ID access to the EJROLE, then connect the ID to the group:

  PERMIT *aa.bbbbbb* CLASS(EJBROLE) ID(*cccc*) ACCESS(READ)

  where:

  - *aa* -- is the security domain identifier for the cell. The value is case sensitive.
  - *bbbbbb* -- is the EJBROLE profile, for example `administrator`. Also case sensitive.
  - *cccc* -- is the **group** ID you wish to grant access

  then:

  CONNECT aaaa GROUP(bbbb)

  where:

  - *aaaa* -- is the user ID
  - *bbbb* -- is the group that you just granted access to the EJBROLE

Exercise: WSADMIN client on a remote workstation

Objective	The WSADMIN client does not need to be on the same MVS image as the Deployment Manager. The options `–conntype SOAP` or `–conntype RMI` allow a network connection to be made across a TCP network. Many people run the WSADMIN client from WebSphere installed on their workstation, and connect to their z/OS-based DMGR. But if security is enabled on the Deployment Manager, it means the workstation client needs to be able to establish the SSL connection. That means the client on the workstation needs to have the CA Certificate that was used to sign the DMGR's server certificate, which is what flows down to the client during the first phases of SSL establishment. Providing the CA Cert to the workstation client involves exporting the cert from RACF, bringing it down to the workstation, and importing it into the TrustFile used by the client there. This exercise will show how it's done.

Error: SSL establishment failure when workstation client doesn't have CA cert

☐ Locate a workstation that has WebSphere installed.

☐ Go to the `\profiles\default\bin` directory where WebSphere is installed.

☐ Issue the following command (all on one line):

```
wsadmin –conntype SOAP –host aaaa –port bbbb
                                         –user cccc –password dddd
```

where:

- *aaaa* is the host address where the z/OS DMGR can be reached
- *bbbb* is the SOAP port of the z/OS DMGR
- *cccc* is the WebSphere Admin ID for the z/OS DMGR
- *dddd* is the password for WebSphere Admin ID for the z/OS DMGR

☐ You should see an error that looks something like this:

```
Error creating "SOAP" connection to host "<host>"; exception information:
com.ibm.websphere.management.exception.ConnectorNotAvailableException:
[SOAPException: faultCode=SOAP-ENV:Client; msg=Error opening socket:
javax.net.ssl.SSLHandshakeException:java.security.cert.CertificateException:
Certificate not Trusted; targetException=java.lang.IllegalArgumentException:
Error opening socket: javax.net.ssl.SSLHandshakeException:
java.security.cert.CertificateException: Certificate not Trusted]
WASX7213I: This scripting client is not connected to a server process;
please refer to the log file wsadmin.traceout for additional information.
WASX8011W: AdminTask object is not available.
WASX7029I: For help, enter: "print Help.help()"
wsadmin>
```

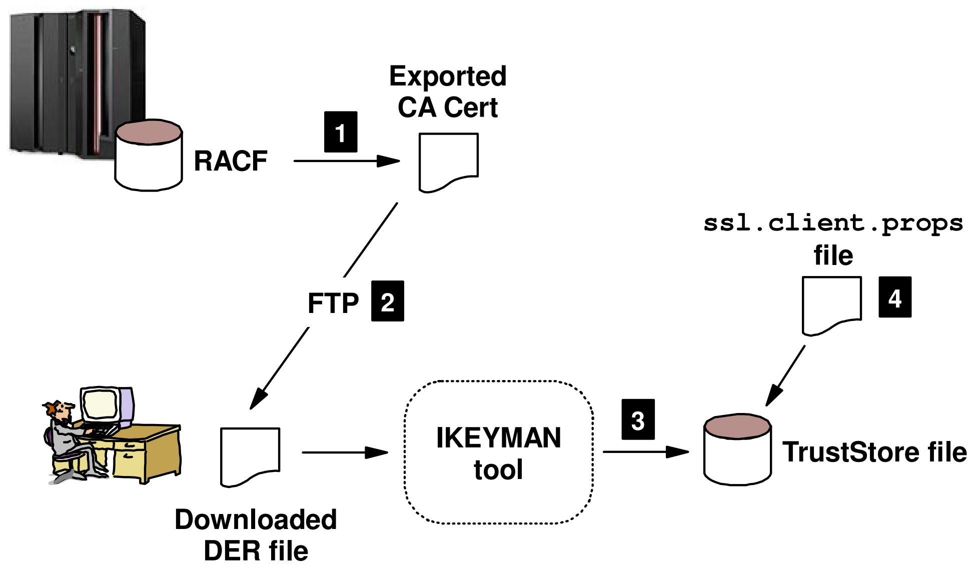
The highlighted text tells the story -- the workstation client received the certificate from the DMGR, but since it did *not* have the CA Cert available it couldn't verify that the "signature" on the certificate could be trusted. Therefore, the SSL connection was disallowed.

The client drops back to a `-conntype NONE` mode, which means it is *not* operating against the DMGR on the z/OS system.

☐ Issue the command **`quit`** at the **`wsadmin>`** prompt.

Background: what's involved with making CA cert available to workstation client

The following picture summarizes the steps involved:



Notes:

1. The CA Cert, which is in RACF on the z/OS system, is first exported to a sequential data set as a "DER encoded PKCS7 certificate chain." The specifics of that is not important ... the key is to get the certificate out of RACF into a format that'll be understood by the IKEYMAN tool. The DER format will give us that.
2. The exported CA Cert is then downloaded in binary format to the workstation where you wish to run the WSADMIN client. The file exists on the workstation as a "DER" file.
3. The IKEYMAN tool is used to create a Trust Store file and the CA Cert brought down from the z/OS system is imported into the new Trust Store.
4. Finally, the `ssl.client.props` file is updated so the WSADMIN client knows to use the new Trust Store file with the imported CA Cert.

Exercise: export CA cert from RACF and download to workstation

☐ Refresh your memory of the exact label on the CA Certificate. See page 89, where you verified it earlier.

☐ Issue the following RACF command (all on one line):

```
RACDCERT CERTAUTH EXPORT(LABEL('aaaa.bb'))
                                        DSN('cccc') FORMAT(CERTDER)
```

where:

- *aaaa* -- is the CA Cert label, something like `WebSphereCA`
- *bb* -- is the Security Domain Identifier, if one was used. Something like `FZ`.
- *cccc* -- is the data set into which you want the exported certificate to go

Note: Check with your security administrator prior to exporting the certificate.

☐ On the workstation, create a directory where you'll store the downloaded certificate and where you'll create the Trust Store file. For example, `C:\wsadmin`

☐ FTP the exported CA certificate data set down to the workstation in *binary* format. Store it in the directory you created for this purpose, and give it an extension of **`der`**.

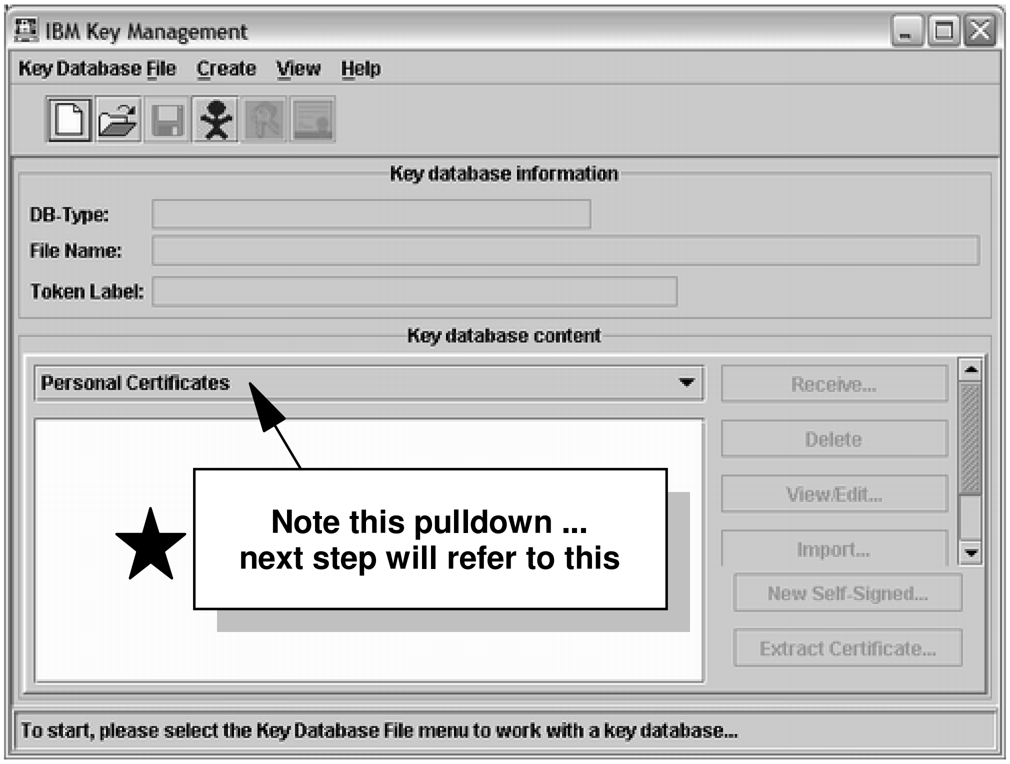
Exercise: use IKEYMAN to create new TrustFile and import CA cert

Important! We're making the assumption that your workstation copy of WebSphere is using the default key and trust store, and that it's okay if we replace them with the trust store we create here. For a personal workstation that's probably okay. For a server running WebSphere that may not be a good assumption. Be sure to back up the properties file we ask you to change.

IKEYMAN is a tool found in the `/profiles/default/bin` directory of the WebSphere configuration structure on the workstation. It provides a way to create the files that store the certificates, and a way to import certificates into the file.

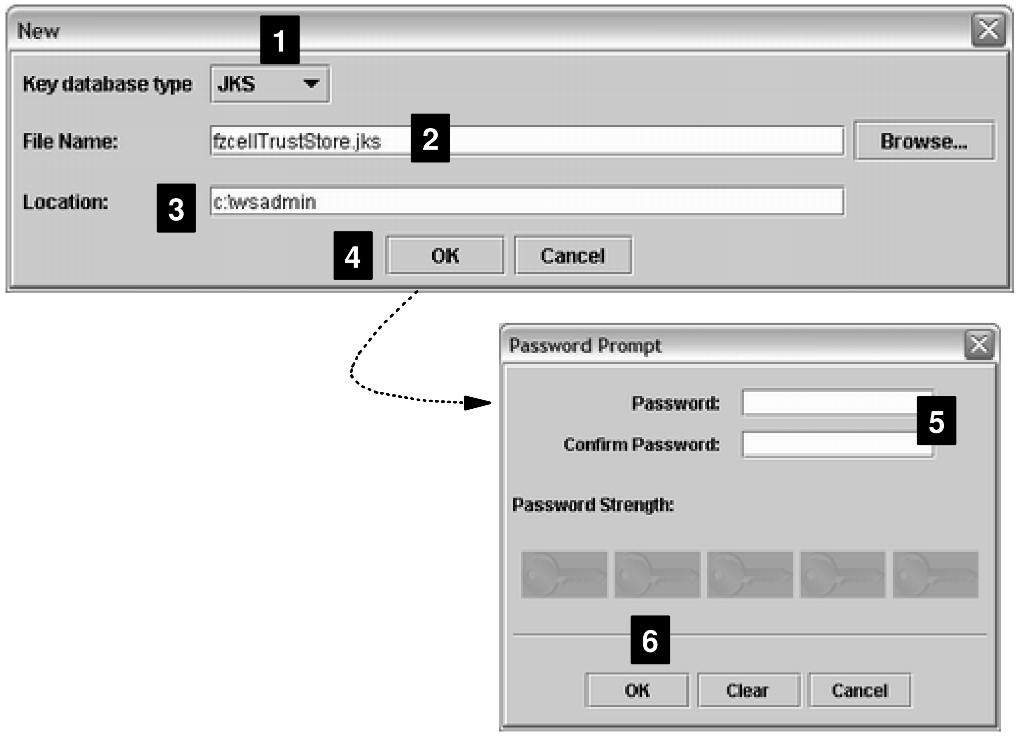
☐ From a command prompt or using Windows Explorer, navigate to the `/profiles/default/bin` directory of the WebSphere configuration on your workstation.

☐ Invoke the `ikeyman.bat` file. You should see a screen that looks like this:



To start, please select the Key Database File menu to work with a key database...

- ☐ Change "Personal Certificates" to ***Signer Certificates***
- ☐ Select ***Key Database File*** ⇨ ***New***
- ☐ Do the following:



Notes:

1. Make sure "JKS" is the database type
2. Provide a name for the keystore. For example: `fzcellTrustStore.jks`, where in `fzcell` is the name of the cell up on z/OS. (The first part of the name is not that important; the `jks` extension is.)

TrustStore File Name:	

3. Specify the location where you want the file created. For example: `c:\wsadmin`. (Location not technically important. Directory must exist; tool won't create it.)

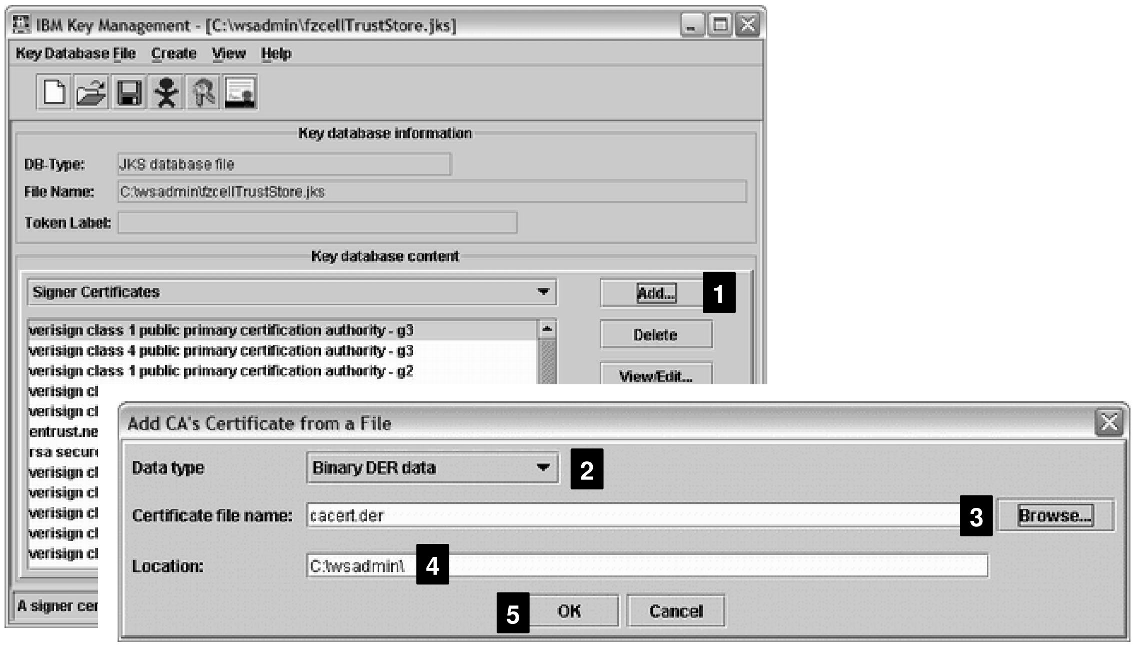
TrustStore Location:	

4. Click on OK
5. Provide a password for the key file. It must be at least six characters long.

TrustStore password:	

6. Click on OK

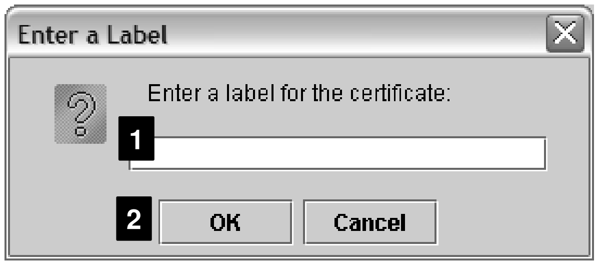
- ☐ We're now ready to import the downloaded certificate file. Do the following:


IBM Key Management - [C:\wsadmin\fzcellTrustStore.jks]
Key Database File Create View Help
Key database information
DB-Type: JKS database file
File Name: C:\wsadmin\fzcellTrustStore.jks
Token Label:
Key database content
Signer Certificates
Add...
1
Delete
View/Edit...
verisign class 1 public primary certification authority - g3
verisign class 4 public primary certification authority - g3
verisign class 1 public primary certification authority - g2
Add CA's Certificate from a File
Data type Binary DER data
2
Certificate file name: cacert.der
3
Browse...
Location: C:\wsadmin\
4
5
OK
Cancel

Notes:

1. Click on the "Add..." button
2. Select "Binary DER data"
3. Use the "Browse..." button to navigate to and select the file you downloaded from the z/OS system and gave an extension of `der`.
4. The "Location" field should be filled in for you based on the selection of your file
5. Click on the "OK" button.

- ☐ Specify the label for the CA Certificate:



Notes:

1. Go back to page 89 and recall the exact name of your Certificate Authority label, including spelling and case. Please that value in this field.
2. Click on the "OK" button.

- ☐ Close the tool
- ☐ Verify the JKS file is indeed in the directory you indicated it should go.

Exercise: update `ssl.client.props` file and `wsadmin.properties` file on workstation

- ☐ On the workstation, navigate to the `/profiles/default/properties` directory in the WebSphere configuration from which you intend to invoke the WSADMIN client.
- ☐ Make ***backup*** copies of your existing `ssl.client.props` and `wsadmin.properties` files
- ☐ Open the original file `ssl.client.props` for edit.
- ☐ Scroll down and find the first instance of `#KeyStore information`, then do the following:

```
  # KeyStore information
  com.ibm.ssl.keyStoreName=ClientDefaultKeyStore
  com.ibm.ssl.keyStore=C:/wsadmin/fzcellTrustStore.jks 1
  com.ibm.ssl.keyStorePassword=fzcell 2
3 #com.ibm.ssl.keyStoreType=PKCS12
  com.ibm.ssl.keyStoreProvider=IBMJCE
  com.ibm.ssl.keyStoreFileBased=true

  # TrustStore information
  com.ibm.ssl.trustStoreName=ClientDefaultTrustStore
  com.ibm.ssl.trustStore=C:/wsadmin/fzcellTrustStore.jks 4
  com.ibm.ssl.trustStorePassword=fzcell 5
6 #com.ibm.ssl.trustStoreType=PKCS12
  com.ibm.ssl.trustStoreProvider=IBMJCE
  com.ibm.ssl.trustStoreFileBased=true
```

Notes:

1. Change the `com.ibm.ssl.keyStore` property to point to the JKS file you just created with IKEYMAN. We asked you to capture the name and location back on page 96. *Note the forward slashes, not backwards slashes Windows would normally use.*
2. Change the `com.ibm.ssl.keyStorePassword` property so it includes the password you provided for your new JKS file. We asked you to capture the password back on page 96.

Note: It is possible to "encode" the password, which makes it not "in the clear" in this file. The `PropFilePasswordEncoder.bat` in the `/profiles/default/bin` directory is used to do this. We're not going to show how to use that utility. It's a little quirky to use. It is invoked with:

`PropFilePasswordEncoder` *`aaaaaa bbbbbb`*

where

- *`aaaaa`* is the location and file name of the properties file you want this utility to go into and encode a password that's initially in the clear
- *`bbbbb`* is the password property within the file that you want this utility to go and encode. For instance, `com.ibm.ssl.keyStorePassword`.

If you have multiple passwords in a properties file that you wish to encode -- the example above has two password properties files -- it means running the utility twice, specifying the same properties file each time, but giving a different property name each time.

The utility also strips out all comments from the properties file. The resulting file looks like it's not the same properties file, but it is. The utility makes a BAK copy. Still, you should make a personal backup copy just to be certain.

3. Comment out the `com.ibm.ssl.keyStoreType=` property. The file you created was not a PKCS12 file.
4. Change the `com.ibm.ssl.trustStore` property to point to the JKS file you just created with IKEYMAN. The same file as for the `keyStore`. We asked you to capture the name and location back on page 96. *Note the forward slashes, not backwards slashes Windows would normally use.*
5. Change the `com.ibm.ssl.trustStorePassword` property so it includes the password you provided for your new JKS file. The same password as for the `keyStore`. We asked you to capture the password back on page 96.
6. Comment out the `com.ibm.ssl.trustStoreType=` property. The file you created was not a PKCS12 file.

☐ Save the file.

☐ Open the original file `wsadmin.properties` for edit.

☐ Change the `com.ibm.ws.scripting.defaultLang=` property so it reads **`jython`**, not `jacl`.

Note: The default script language is determined by the *client*, not the server. That's defined in the `wsadmin.properties` file. Since the client here is now on the workstation, the `wsadmin.properties` file on the workstation is what takes effect.

☐ Save the file.

Exercise: invoke workstation WSADMIN client and connect to DMGR on z/OS

☐ On your workstation, navigate to the /profiles/default/bin directory and issue the following command (all on one line):

```
wsadmin –conntype SOAP –host aaaa –port bbbb
                                        –user cccc –password dddd
```

where:

- *aaaa* is the host address where the z/OS DMGR can be reached
- *bbbb* is the SOAP port of the z/OS DMGR
- *cccc* is the WebSphere Admin ID for the z/OS DMGR
- *dddd* is the password for WebSphere Admin ID for the z/OS DMGR

☐ You should see *something* like this:

```
WASX7209I: Connected to process "dmgr" on node <node> using SOAP
connector;  The type of process is: DeploymentManager

WASX7031I: For help, enter: "print Help.help()"

wsadmin>
```

Note: Look carefully to make sure you actually established the session. An error might cause it to drop back to a `–conntype NONE` mode. The next step is also one way to verify you are actually connected to the z/OS DMGR.

☐ Issue the following command:

```
print AdminApp.list()
```

The list of applications you get back should be those deployed on the cell up on z/OS.

Review	The point of this exercise was to show how the WSADMIN client on the workstation. This assumed you had WebSphere Application Server on your workstation, and thus the `wsadmin.bat` file there. Invoking `wsadmin.bat` is easy; getting the SSL session established is what this exercise was all about. To accomplish this, we exported the CA certificate from z/OS RACF and brought it down to the workstation, then used IKEYMAN to create a new trust store file with that CA cert. We then updated the `ssl.client.props` file to point to the new trust store file. That's what allowed the WSADMIN client on the workstation to accept the DMGR's server certificate and build the SSL session.

Topic: fine grained security

Fine grained security is a new feature of V6.1 which allows the scope of authority for WSADMIN actions to be limited based on the userid running WSADMIN. In other words, if you have a user, "Fred," who you want to be able to install applications into a certain node, but do nothing else in the cell, you can now do that.

Note: This section borrows heavily from Mike Loos' paper on Techdocs:

`http://www.ibm.com/support/techdocs/atsmastr.nsf/WebIndex/`**`TD103324`**

I encourage you to pull that paper if you want more details on the topic of fine grained security and WSADMIN.

Background: evolution of access authority in WebSphere z/OS

Originally, WebSphere had only one level of access authority: if a user was allowed in the front door, they could have full administrator rights.

Somewhere back in the V5.x days they addressed this by creating four levels of administrative authority:

- *Administrator* -- full authority. The Admin ID was by default granted access to this.
- *Monitor* -- ability to look at the configuration, but not change anything
- *Configurator* -- ability to modify the configuration, such as install applications and add servers.
- *Operator* -- ability to start and stop servers, but little else.

In the z/OS world this access authority was controlled through the use of `EJBROLE` profiles in SAF. Four `EJBROLE` profiles with these names were created, and if an ID was granted `READ` to the role it had that authority. By default the Admin ID was granted `READ` to the administrator role.

Dividing the access authority into four levels was good, but not perfect. Two shortcomings existed: first, many customers wanted a role that allowed someone to deploy applications, but nothing else. That role as "deployer" was seen as a needed role. Second, the `EJBROLE` profiles were for the *entire cell*. If someone was given authority to be a `configurator` for the cell `MYCELL`, for example, they would be a `configurator` for *any part of the cell*. There was a desire to limit that authority so a user's access could be limited to some portion of a cell.

In Version 6.1 they've addressed both issues:

- A new `EJBROLE` has been created called `deployer`. That limits a user's authority to just deploying and removing applications.

  Note: There's also a new role called `adminsecuritymanager`, which has the ability to turn security on or off for a cell, and also is what gives an ID the authority to define all this fine grained security definitions. The Admin ID has access to this by default.

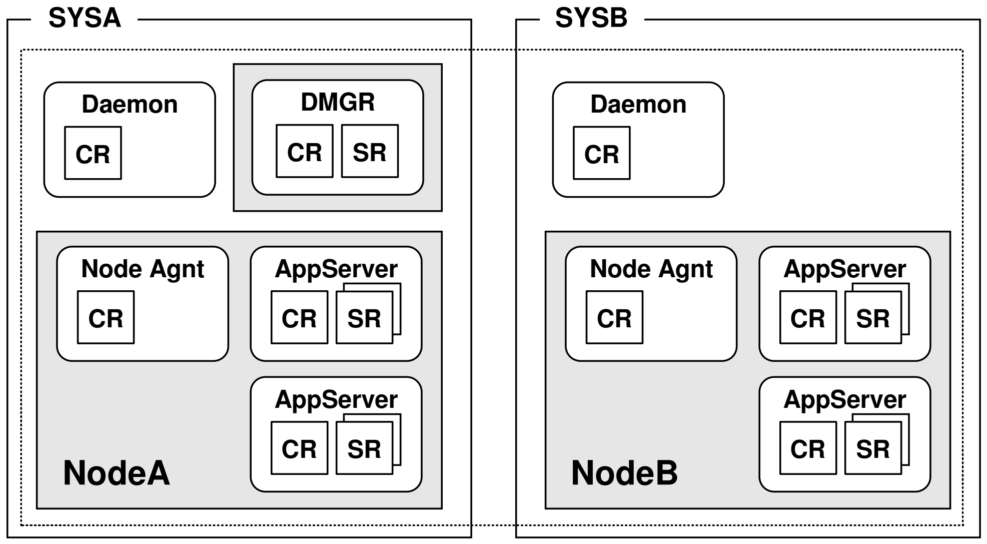
- The "fine grained security" function was added, which provided a way to define what part of a cell a given user could operate in. That user would have the authority of the `EJBROLE` they were given access to, and in addition the extent of their authority could be defined as some subset of the entire cell.

Note: Be aware that "fine grained security" applies only to WSADMIN access, not console.

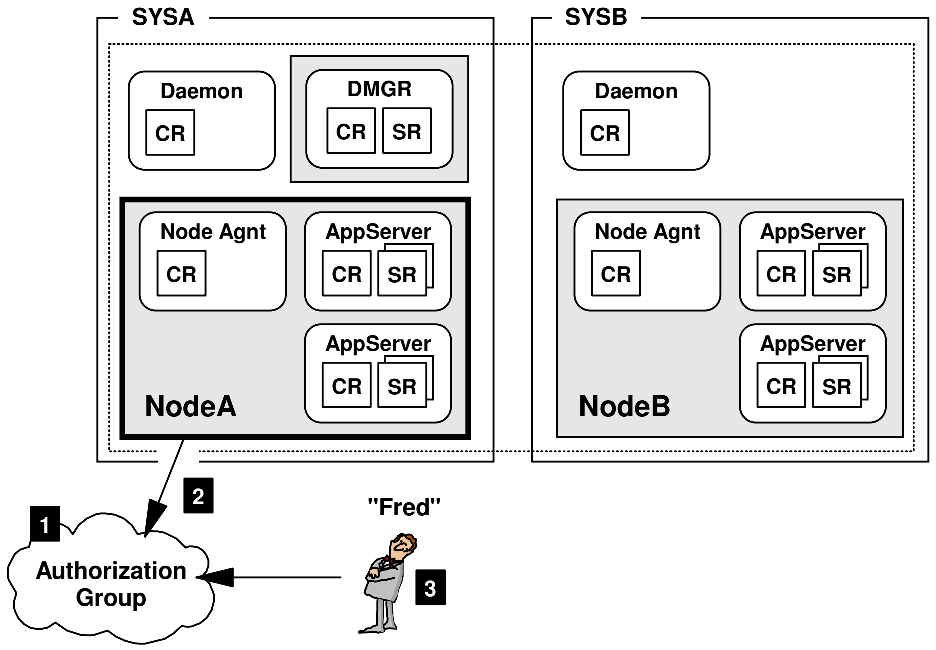
When you first set up a V6.1 cell, there is no "fine grained security" in effect ... the Admin ID is granted access to both `administrator` and `adminsecuritymanager`, and there's nothing defined initially to limit a user's access to some portion of the cell. It's the whole cell initially, just as before.

Background: how fine grained security works, at a high level

Imagine a cell that looked like this ... a typical two-LPAR cell:



Now imagine further that you have a userid called `FRED` you will use for WSADMIN, and you want to limit that ID's authority so it can only do the `deployer` role, and you want to limit it to only being able to do deployer role things in `NODEA`. Accomplishing that with fine grained security involves the following:



Notes:

Note: Do **not** execute these commands just yet. There's just a little more to it then we show here. Try to get a feel for the *concept* here.

1. Creating an "Authorization Group" definition using the WSADMIN `AdminTask` object. For instance, the command would be:

   ```
   AdminTask.createAuthorizationGroup('[-authorizationGroupName Dep_NodeA]')
   ```

   It's really little more than a collection point where the userid and resource can be associated with one another.

2. A part of the cell is now assigned to the authorization group you just created::

   ```
   AdminTask.addResourceToAuthorizationGroup('[
                          -authorizationGroupName Dep_NodeA -resourceName Node=NodeA]')
   ```

   In this example the node `NodeA` is assigned to the authorization group.

3. The final piece of the puzzle is to assign `FRED` to the authorization group, and indicate what "role" he has:

   ```
   AdminTask.mapUsersToAdminRole('[
                -authorizationGroupName Dep_NodeA -roleName deployer -userids FRED]')
   ```

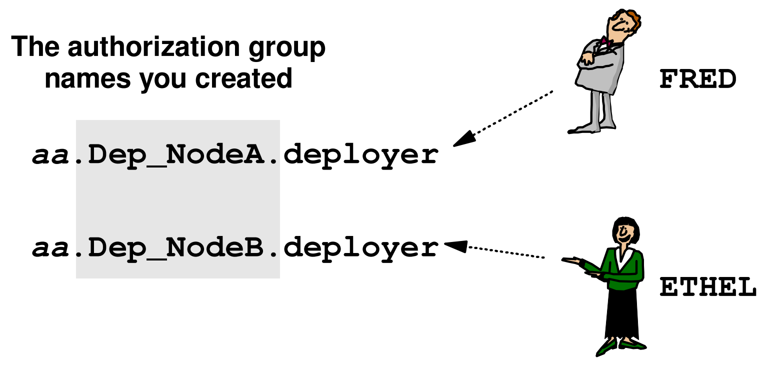
   In other words, "Grant `FRED` access to the authorization group named `Dep_NodeA` and give him the `deployer` role." The scope of his access is limited to only those resources that have been added to the authorization group. So far only NodeA has been added. So `FRED` has deployer authority to `NodeA` only.

If you had someone else -- `ETHEL` -- who you wanted to be a deployer for just `NodeB`, you'd go through a similar exercise:

- Create a second authorization group ... say `Dep_NodeB`.
- Assign `NodeB` to that authorization group
- Map `ETHEL` to that authorization group with the role `deployer`.

In essence what you've done is you've tied `FRED` as `deployer` to `NodeA` by using the authorization group `Dep_NodeA` as the place where `FRED` and `NodeA` are joined. The same holds for `ETHEL`, but using a different authorization group.

But that's not quite the whole story. There would also be two additional RACF `EJBROLE` profiles, and `READ` access granted to those profiles:



And the picture is complete: when `FRED` tries to do something in WSADMIN, his ID is checked to see if it's part of an authorization group. If so, then WebSphere checks RACF to make sure `FRED` has the proper access to the associated `EJBROLE`. If it all checks out, he's allowed to operate within the scope of the resources assigned to the authorization group, and do only those things permitted by the role he has -- `deployer` in this example.

Background: who is allowed to configure and manage the fine grain definitions

Only an ID that has access to the `adminsecuritymanager` EJBROLE has the authority to issue the AdminTask commands needed to setup this fine grained security environment. By default that means the WebSphere Admin ID has that authority because at cell creation that ID was granted `READ` to that EJBROLE. Other IDs can be added, but by default the Admin ID has the authority from the start.

Overview: fine grain security exercises

We're about to launch into a series of exercises on implementing fine grain security. This is a challenge because the exercises rely on things being in place at your location:

- A WebSphere cell with at least two application servers
- An ID to use that is *not* the WebSphere Admin ID. This implies you've worked through the exercises under "Exercise: use a different ID from WebSphere Admin ID" starting on page 85. In these exercises we'll use the ID `FRED` as an example.

What we'll do is create a single authorization group and assign *one* of the application servers to it. Then we'll grant `FRED` deployer access to it. But we won't grant `FRED` access to the other server. We'll test this by having `FRED` use WSADMIN to install SuperSnoop into one application server then the other. It'll work for the one `FRED` has access to; it'll be rejected for the server `FRED` does not have access to.

Exercise: create authorization group and assign resource and ID to it

Objective	To create the definitions within WebSphere to enable fine grained security for the hypothetical ID `FRED`.

- ☐ Using the *WebSphere Admin ID* start a WSADMIN session by connecting to the Deployment Manager.

  Note: Using the WAS Admin ID is critically important. That ID is by default granted access to the `adminsecuritymanager` EJBROLE. Access to that is required to issue the `AdminTask` commands that establish the fine grained security environment.

- ☐ At the WSADMIN command prompt, issue the following command:

  ```
  print AdminTask.listAuthorizationGroups()
  ```

  You should see nothing in return ... you have no authorization groups yet.

- ☐ Create a file called `fg1.jy` and include the following:

  Note: This file supplied in the ZIP that accompanies the Techdoc.

```
  # ------------------------------------------------------------------
  # Set basic variables
  #  resource_type = 'Server' 'Node' 'Cluster' 'Cell' 'Application'
  #  role_name = 'administrator' 'configurator' 'operator'
  #              'monitor' or 'deployer'
  #  !! case matters -- type carefully !!
  # ------------------------------------------------------------------
  group_name    = 'aaaa'
  resource_type = 'bbbb'
  resource_name = 'cccc' 1
  role_name     = 'dddd'
  userid        = 'eeee'
  # ------------------------------------------------------------------
  # Construct commands and invoke
  # ------------------------------------------------------------------
2 AdminTask.createAuthorizationGroup\
   ('[-authorizationGroupName ' + group_name + ']')
  print 'Created authorization group'
3 AdminTask.addResourceToAuthorizationGroup\
   ('[-authorizationGroupName ' + group_name +\
    ' -resourceName ' + resource_type + '=' + resource_name + ']')
  print 'Add resource to group'
4 AdminTask.mapUsersToAdminRole\
   ('[-authorizationGroupName ' + group_name +\
    ' -roleName ' + role_name +\
    ' -userids ' + userid + ']')
  print 'Mapped user to group'
5 AdminConfig.save()
  print 'Saved configuration changes'
  # ------------------------------------------------------------------
  # Perform a 'refreshAll'
  # ------------------------------------------------------------------
  dmgr_id = AdminControl.queryNames\
6   ('WebSphere:type=AuthorizationGroupManager,process=dmgr,*')
  AdminControl.invoke(dmgr_id,'refreshAll')
  print 'Performed refreshAll'
```

Notes:

1. Variables set:
   - `aaaa` -- is the authorization group name you wish to create. For instance, in our example we used `TestGroup`.
   - `bbbb` -- the resource type. Options shown in the comments at the top of the script. There may be others, but those are the most common ones. We set this to `Server` for our example.
   - `cccc` -- the resource name. This is related to the resource type you just set. This would be the long name of the resource. For instance, we set the type as `Server`. We set the name as `fzsr01c`.
   - `dddd` -- the role name. Options shown in the comments at the top. For our example we set this to `deployer`.
   - `eeee` -- the userid we're granting access to. In our example, `FRED`.
2. The authorization group is created
3. The resource is added to the authorization group
4. The user is mapped to the authorization group
5. The configuration changes are saved
6. We perform a `refreshAll`, which is like tapping the DMGR on the shoulder and telling it, "You have changes, be aware of them." Apparently this requires more than just a configuration save.

☐ Save the file and FTP it to your system. Make sure the permissions allow WSADMIN to read the file. Then issue the command:

```
execfile('/aaaa/fg1.jy')
```

Where `aaaa` is the location of your script file. If things work out well, you should see:

```
Created authorization group
Add resource to group
Mapped user to group
Saved configuration changes
Performed refreshAll
wsadmin>
```

☐ At the WSADMIN command prompt, issue the following command again:

```
print AdminTask.listAuthorizationGroups()
```

It should report back with the single authorization group you've created.

Review	With one script you've created an authorization group, assigned a resource to it and then assigned a userid as well. We're almost ready to test this. But first, some RACF work needs to be done.

Exercise: create RACF EJBROLE

Objective	To see how the underlying RACF profiles are built that support of the fine grain security structure.

Note: Review these proposed changes with your security administrator.

☐ Go back to page 92 and note the security domain identifier for your cell, if in fact one was used.

☐ Issue the following RACF command:

```
RDEFINE EJBROLE aa.bbbbbb.cccccccc UACC(NONE)
```

where:

- *aa* -- is the "security domain identifier" for your cell. If your cell has no security domain identifier, omit this, including the dot that follows. Case matters.
- *bbbbb* -- is the authorization group name you created in the script you just ran. For example, we suggested our example was `TestGroup`. Case matters.
- *ccccccccc* -- is the role you granted the ID in the script you just ran. For example, we suggested our example was `deployer`. Case matters.

For example: `RDEFINE EJBROLE FZ.TestGroup.deployer UACC(NONE)`

☐ Now issue the following RACF command:

```
PERMIT aa.bbbbbb.cccccccc CLASS(EJBROLE) ID(dddddd) ACCESS(READ)
```

where:

- *aa*, *bbbbbb* and *cccccccc* are the exact same values you used for the `RDEFINE` command.
- *dddddd* is the userid. `FRED` was the example we were using.

For example:

```
PERMIT FZ.TestGroup.deployer CLASS(EJBROLE) ID(FRED) ACCESS(READ)
```

☐ Finally, issue the following:

```
SETROPTS RACLIST(EJBROLE) REFRESH
```

Review	You've created the RACF underpinnings to the authorization group definitions you created with the `fg1.jy` script.

Exercise: test fine grained security

Objective	To show that fine grain security is working. We'll do this in a very simple way: we'll try to install SuperSnoop into the server our ID does not have authority for. We granted `FRED` authority to deploy into `fzsr01c`. We'll try to install into `fzsr02c`.

Note: You may want to shut down your entire WebSphere environment and restart it. It's not necessarily required, but WebSphere does various caching of RACF stuff and some of what you've done may or may not be in the cache. If you recycle your environment (which we hope is a test environment and not your production environment), you'll be fairly certain of a nice, clean test.

☐ Start a WSADMIN command prompt session using `-conntype SOAP` or `RMI` and pass in the parameters `-user` and `-password` for the ID you're working with for this fine grain security exercise. (It is `FRED` for us.)

☐ First, install SuperSnoop into the server you believe the ID *does* have authority. This will validate the syntax of the installation command. Issue the command:

```
AdminApp.install('/aaaa/SuperSnoopProj.ear','[-node bbbb -server cccc]')
```

where:

- *`aaaa`* -- is the location where you have the SuperSnoopProj.ear file stored.
- *`bbbb`* -- is the long name of the node
- *`cccc`* -- is the long name of the server the ID does have authority to deploy into

You should see something like this:

```
ADMA5013I: Application SuperSnoop installed successfully.
```

Note: You've verified that your new ID has at least `deployer` for the one server it should.

☐ Now *uninstall* SuperSnoop:

```
AdminApp.uninstall('SuperSnoop')
```

☐ And tidy things up:

```
AdminConfig.save()
```

☐ Next, install SuperSnoop into the server you believe the ID does *not* have authority. This will validate fine grain security. Issue the command:

```
AdminApp.install('/aaaa/SuperSnoopProj.ear','[-node bbbb -server cccc]')
```

where:

- *`aaaa`* -- is the location where you have the SuperSnoopProj.ear file stored.
- *`bbbb`* -- is the long name of the node
- *`cccc`* -- is the long name of the server the ID does *not* have authority to deploy into

You should see *something* like this:

```
ADMA0187E: Authorization failure. The user does not have permission
to install application cells/fzcell/nodes/fznodec/servers/fzsr02c.
The user must have deployer role or configurator role for the cell or
for the targets to install/redeploy the application.
```

Note: Fine grained security worked. In our case, `FRED` could not install into server `fzsr02c`, which is what we expected.

Review	Fine grain security only applies to the WSADMIN environment; not to the Admin Console environment. It involves creating authorization groups using WSADMIN, assigning a resource (or resources) to that group, then mapping users to the group. This establishes the relationship that defines who is allowed to access what resources and what authority they have. It's an involved process. If you're comfortable having all your WSADMIN work done under the authority of your Admin ID (which has administrator), then continue with that. But if you have people you wish to restrict in their scope and authority, then fine grained security provides it. Again, for another treatment of this topic, see: `http://www.ibm.com/support/techdocs/atsmastr.nsf/WebIndex/`**`TD103324`**

Lesson wrap-up and summary

In this lesson we've explored three aspects of security and WSADMIN

- Using an ID other than the WebSphere Admin ID
- Invoking WSADMIN from a distributed platform and connecting the DMGR on z/OS
- Implementing "fine grained security" to restrict a user to a subset of the entire cell

Security is a very complex topic. Always consult with your security administrator before making security changes to your environment.

Suggestions For Improvement?

If you believe there is some aspect of WSADMIN not represented in this document, please send an e-mail to:

`dbagwell@us.ibm.com`

and I'll consider it for update. I'm particularly interested in things related to application installation that you believe will benefit others. The goal of this document is to introduce the reader to key concepts and essential things, so let's not go too far from that objective. (In other words, let's not go too deep into esoteric installation things.)

A near-term objective is to augment this document at some point with an appendix that contains a library of useful Jython code examples. We've already provided a few in this document -- node synchronization, application installation, cluster creation, port remapping. Mike Loos' white paper on Jython scripting (WP100963) has other interesting stuff, such as changing a servant region's JVM heap sizes. Suggestions as to what sorts of things might be good to show in such an appendix is appreciated.

Document Change History

Check the date in the footer of the document for the version of the document.

May 07, 2007	Original document.
May 08, 2007	Added the number WP101014 to the document. Also added a URL reference to a tool that will assist in converting JACL scripts to Jython.
September 10, 2008	Added a section on "Further Exploration of the AdminTask Object." This was in response to the discovery that some AdminTask commands were not in the InfoCenter and it was requested they be documented here. A document APAR is now opened and the InfoCenter will be updated. But adding the information here also gave us the opportunity to show how a listing of the "command groups" is possible.

End of Document WP101014