WebSphere Application Server for z/OS Version 5.0.2

WSADMIN Scripting Primer

***Preliminary Release** -- document not yet indexed.*
Look for update in future with index.

This document can be found on the web at:
`www.ibm.com/support/techdocs`
Search for document number **`WP100421`** under the category of "White Papers"

Version Date: May 5, 2004

IBM Washington Systems Center

Carl Wohlers
IBM WebSphere for zSeries Sales
1-919-847-1966
`carlw@us.ibm.com`

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

Thanks to:

- **Mike Cox** of the Washington Systems Center
- **Jack Brady** of WebSphere Development
- **Rohith Ashok** of WebSphere Development
- **Nick Carlin** of IBM UK

Suggestions for Future Updates:

If you have some suggestions for how to improve this document -- things that need further clarification, or WSADMIN examples that need to be included -- please send them to:

dbagwell@us.ibm.com

Table of Contents

Summary Overview

WSADMIN is a scripting interface into WebSphere Application Server that permits the automation of many different tasks. Starting with Version 5 of the product, the scripting interface is now common across all platforms: zSeries, pSeries, xSeries and iSeries.

Generally speaking, WSADMIN is most powerful when used to automate frequently executed tasks, such as installing applications. There the objective is often to remove as much potential for inconsistency from the process as possible. The user-interface for WebSphere Application Server -- the "Admin Console" -- is a standard point-and-click web interface. It is quite powerful, but the potential is there to do things differently on environment A as compared to B.

Why this document was written

WSADMIN is a fairly complex scripting interface, and can be difficult to approach for someone with little or no knowledge of the topic. The WebSphere "InfoCenter" web site is a rich source of WSADMIN examples, but unless someone has a working knowledge of WSADMIN, those examples can be quite intimidating.

What this document is intended to provide -- and not provide

The objective of this document is to provide a basic working knowledge of WSADMIN so that the InfoCenter's examples can be used to greater advantage. This document makes no attempt to be a complete reference for WSADMIN command syntax; the InfoCenter serves that function. Also, this exercises in this document are limited to relatively simple tasks. We believe keeping things simple will allow the primary objective to best be met: a basic working knowledge of WSADMIN.

How this document is designed

This document is designed to be a "primer." Readers are introduced to important concepts in a systematic way, and knowledge is built as the reader progresses through the document. The document provides a series of step-by-step exercises to reinforce the concepts. The document is intended to be worked through from front to back, though readers with some preliminary WSADMIN knowledge may skip ahead as they see appropriate.

Note: The exercises are provided as separate text files or, if you don't have the files, inline in this document under "Appendix A: Exercises (available for copy-and-paste)" starting on page 94. Use Acrobat's "text selection" function to extract.

The z/OS focus of this document

As mentioned, WSADMIN is common across all platforms on which WebSphere Application Server Version 5 runs. The command syntax is essentially the same across all platforms. But the *manner* in which the commands are executed may differ slightly. WebSphere Application Server for z/OS also has certain characteristics unique to the platform, such as: "short name" values; server processes comprised of controller and servant; the need to coordinate MVS definitions such as RACF and WLM to the WebSphere structure.

This document maintains a focus on the z/OS platform, though much of the knowledge gained on z/OS can be carried to other platforms as well.

Where reference documentation is available

The WebSphere Application Server "InfoCenter" is located at:

`http://publib.boulder.ibm.com/infocenter/wasinfo/index.jsp`

You may then search on specific keywords, or search on "WSADMIN examples" and receive a long list of common WSADMIN tasks. Or use the navigation panel to go to:

Reference ⇨ Scripting Interfaces

Introduction to WSADMIN

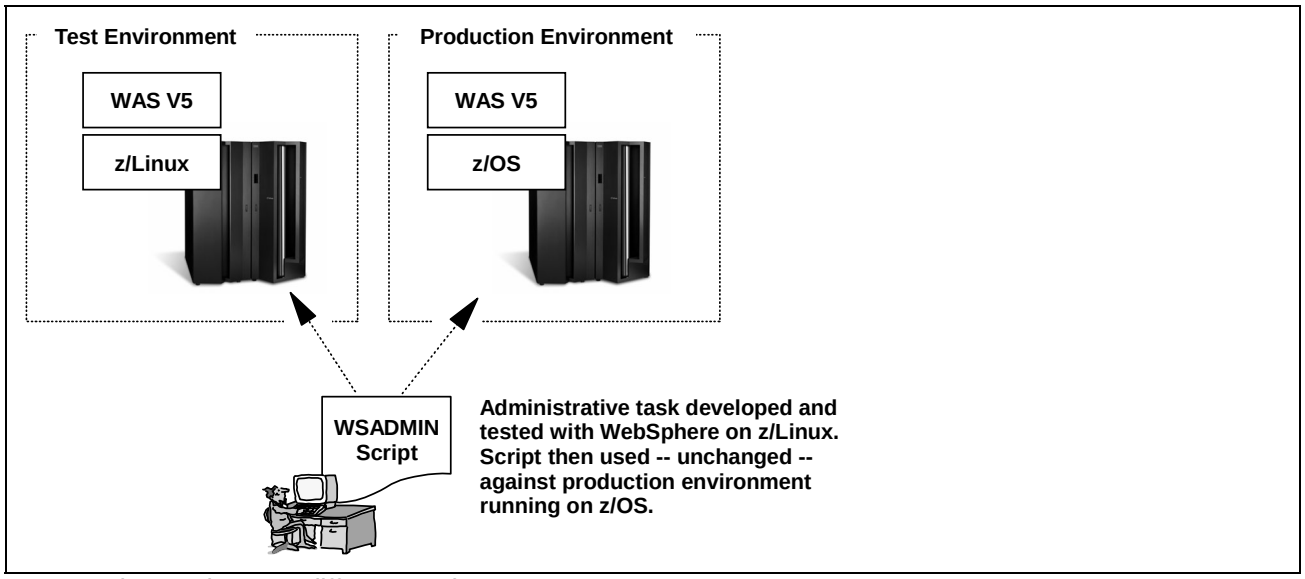
What is WSADMIN? The most basic answer to that question is this: WSADMIN is a feature of WebSphere Application Server that allows a program to do the things a human operator does when they point-and-click on the Administrative Console. So in that sense, WSADMIN is a feature that provides a way to *automate* administrative tasks. It does this by providing a *scripting interface* into the administrative function of WebSphere.

Note: It's important to understand at this point what WSADMIN is *not*. It is not a way to simply record and playback the mouse movements, mouse clicks and keyboard entry of someone sitting at the Admin Console of WebSphere. What the *scripting interface* provides is a way for another program to "connect to" WebSphere Application Server and accomplish administrative tasks by issuing commands with keywords, options and parameters. It does this by providing four Java "objects" -- `AdminApp`, `AdminConfig`, `AdminControl` and `Help`. Each object has a number of "methods" that may be invoked by your program (called a "script").

Understanding what all the methods are, how to code the syntax for the methods, and knowing what values to supply for the parameters is the challenge in learning WSADMIN.

The WSADMIN scripting interface is common across all the platforms on which WebSphere Application Server Version 5 runs. This is different than in Version 4, where at one point in time the distributed platforms migrated up to WSADMIN while the z/OS platform continued to use the older interface.

WSADMIN being universal across the platforms is a good thing: it allows the same script to be used in different environments. So, for example, a script written for a test environment running on *z/Linux* may be reused against a production *z/OS* server.

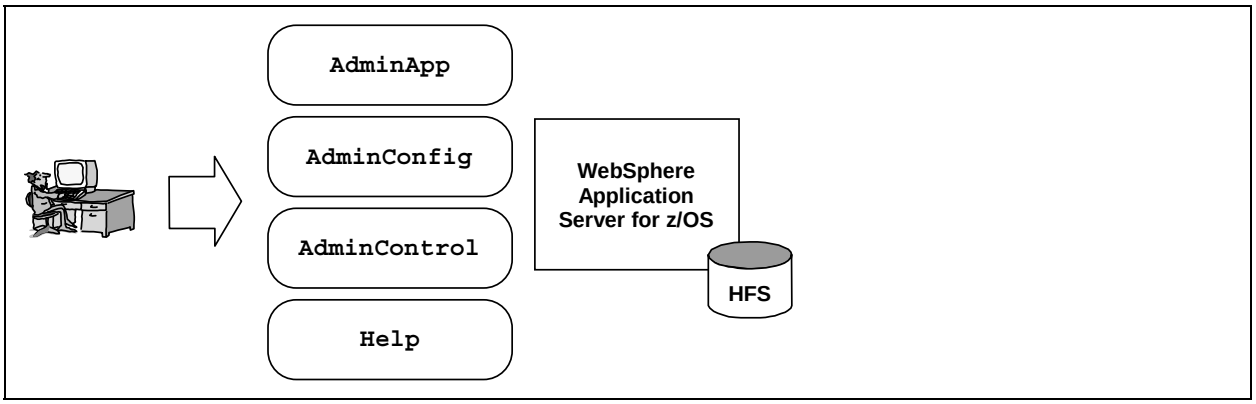


Same script used across different environments

What WSADMIN is not

It's important to understand at this point what WSADMIN is not. It is not a way to simply record and playback the mouse movements, mouse clicks and keyboard entry of someone sitting at the Admin Console of WebSphere. What the scripting interface provides is a way for another program to accomplish administrative tasks by issuing commands with keywords, options and parameters.

It does this by providing four Java "objects" -- `AdminApp`, `AdminConfig`, `AdminControl` and `Help`. Each object has a number of "methods" that may be invoked by your program (called a "script"):



Four Java "objects" provided with WSADMIN; your script operates against methods on the objects

If you look at the table of contents for this document, you'll see that it's organized around the first three of these objects.

If you've configured a WebSphere for z/OS cell, you've used WSADMIN

Yes, you have: the `BBOWIAPP` batch job (for a Base Application Server node) and the `BBODIAPP` batch job (for a Deployment Manager) both invoked WSADMIN to install the Administrative application into the server. WSADMIN was invoked out of the JCL, and the script used to install the application was included in the JCL. We explore this under "A look inside the BBODIAPP job" on page 21.

The wsadmin.sh shell script

The way in which WSADMIN is invoked is with the `wsadmin.sh` shell script. In a Network Deployment configuration, you will have several different copies of that shell script in the HFS:

- in the `/bin` directory of the Deployment Manager
- in the `/bin` directory of each Node Agent
- in the `/bin` directory of each application server

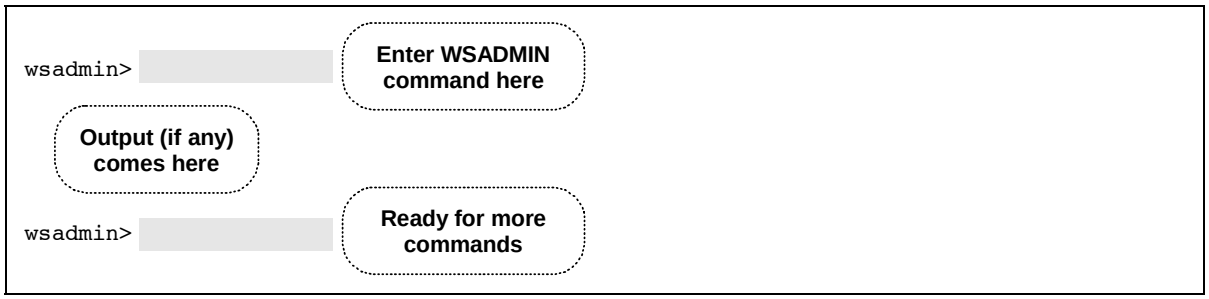
??? Why three copies? Each of those server types -- Deployment Manager, Node Agent and application server -- is a "managed process," and WSADMIN may connect to each. By locating a copy of `wsadmin.sh` in the `/bin` directory of each server, it allows WSADMIN to have a default managed process to act upon. If you invoke `wsadmin.sh` in the `/bin` of the Deployment Manager, for example, and supply no parameters telling it to connect somewhere else, it'll default and connect to the Deployment Manager. We'll talk about "connecting" a bit later.

Note: WebSphere Application Server for z/OS uses a shell script, as do the Unix and Linux products. The xSeries product uses a batch (`*.bat`) file. They all accomplish the same thing.

Two ways to use WSADMIN -- interactive and batch

The difference between the two lies in how the commands are passed to WSADMIN:

- In *interactive* mode you have a command entry prompt. You enter the WSADMIN commands at the prompt. WSADMIN accepts the commands, processes them, then presents the command prompt again:



Entering WSADMIN commands at the `wsadmin>` command prompt (interactive mode)

- In *batch* mode you supply the commands to WSADMIN as a block of commands, either in a separate file or as a block of "inline" commands.

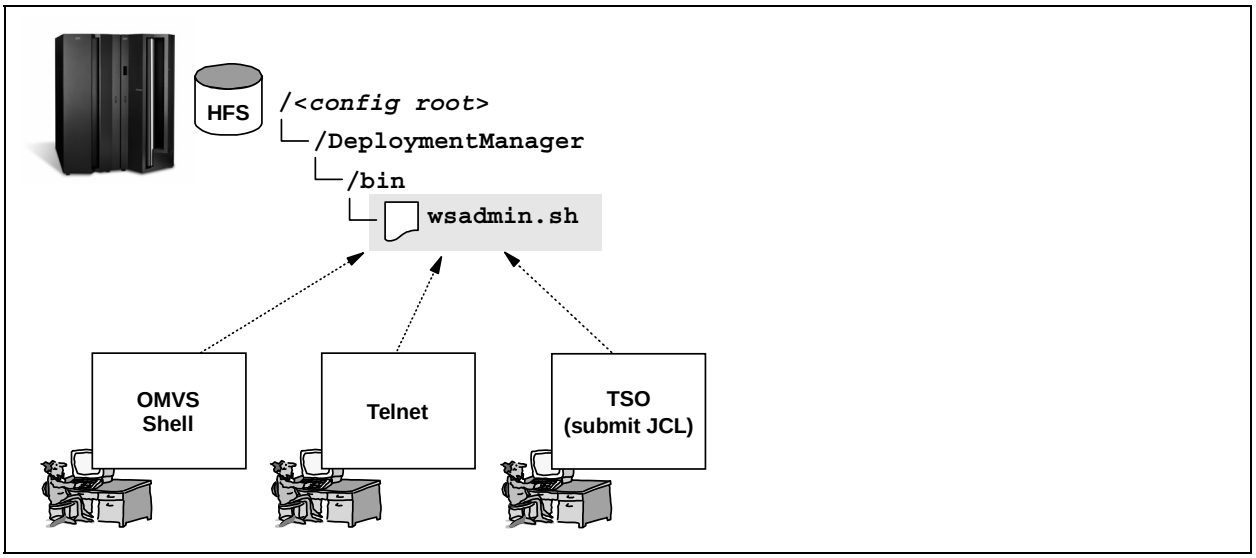
WSADMIN commands supplied in external file, or strung along "inline" with shell script invocation

Note: The "inline" method is what was used in the `BBOWIAPP` job to install the Admin application.

Interactive mode is nice when the commands being entered are relatively short, and you're not looking to save the commands for re-use later. You'll see the use of interactive mode throughout this document when we're querying the "help" method of WSADMIN, and when we're doing simple things like listing out installed applications. Batch mode is nice when the commands being entered are long and complex, and when they'll be used later. An example of the use of batch mode would be a set of commands that installs an application.

Invoking the `wsadmin.sh` shell script from OMVS, Telnet, or JCL

Sitting at your computer, you have a few different ways in which you can invoke the `wsadmin.sh` shell script:

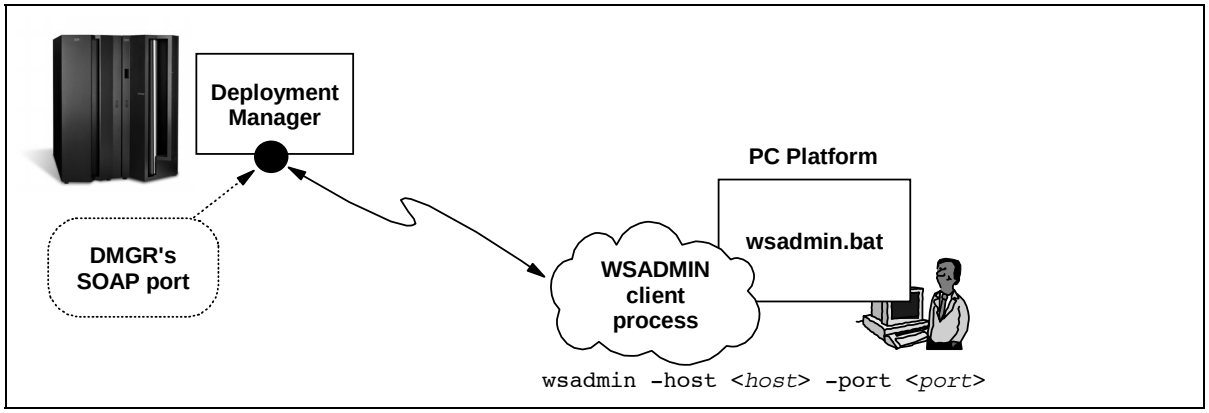


Three different ways to invoke `wsadmin.sh`

Which you use is really a matter of personal preference. In this document we most often instruct the use of the OMVS shell, but in truth a Telnet terminal would work perfectly well. Using a JCL job to invoke WSADMIN may also be used, and we illustrate that method in several places in the document.

A fourth way to invoke WSADMIN -- on another platform

WSADMIN can be instructed to connect to a running server process, and in fact we'll do that quite a bit in this document. You pass in a `-host` and `-port` parameter to accomplish this. That means that WSADMIN on *any* server platform can connect to another server:



Invoking WSADMIN on a PC and connecting across the network to WebSphere on z/OS

Note: The key point is that when WSADMIN is told to connect to a server process, that server process may be on any system on the network. But this method can get complicated, particularly when SSL will be used: it requires the coordination of keyrings on both systems. *For this document we'll avoid this and rely on OMVS, Telnet or JCL batch submission.*

Important: run `wsadmin.sh` under authority of WebSphere Administrator ID

The "WebSphere Administrator ID" is one of the RACF IDs created when you created your WebSphere configuration. It was defined in the "Security Domain" phase of configuration. The default value is `WSADMIN`, though your value will likely be something different.

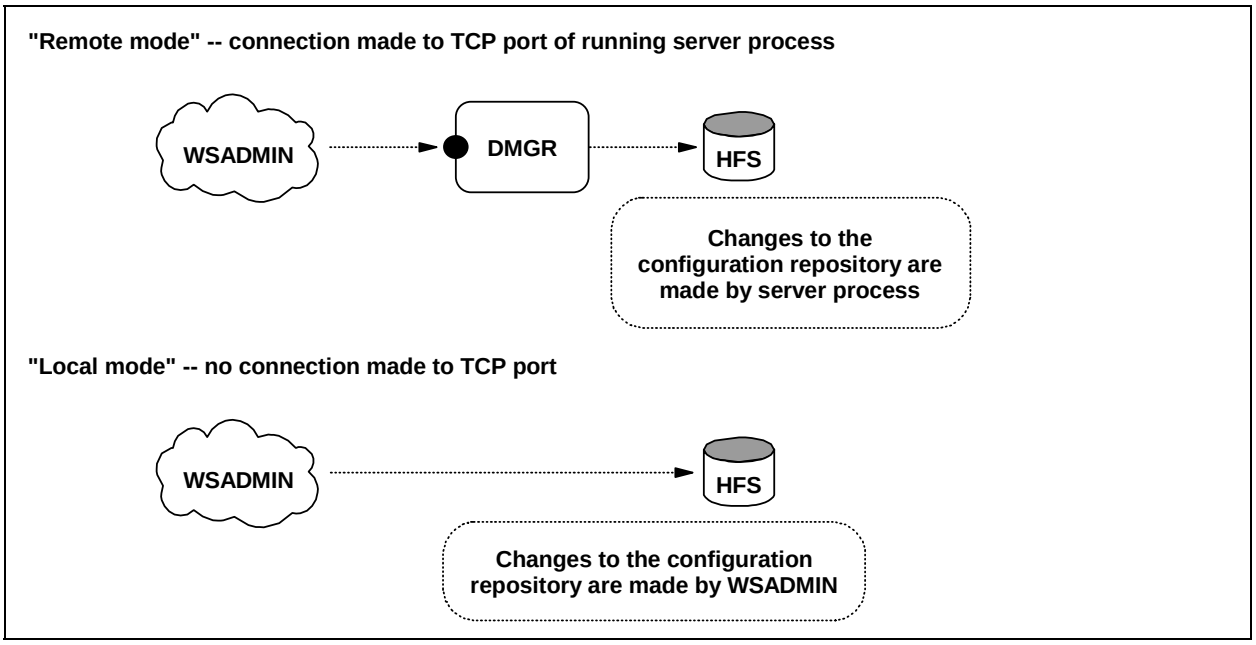
When running the `wsadmin.sh` shell script, it's important to have the script run under the authority of this userid. That will give WSADMIN the proper permissions to read, write and search the configuration HFS directory.

There are two ways to run the shell script under the authority of that ID:

OMVS or Telnet	Before invoking `wsadmin.sh`, "switch users" (`su`) to the WebSphere Administrator ID. Provide the password. Then invoke `wsadmin.sh`
JCL batch	Make sure that the `JOB` card has the `USER=` and `PASSWORD=` values for the WebSphere Administrator ID.

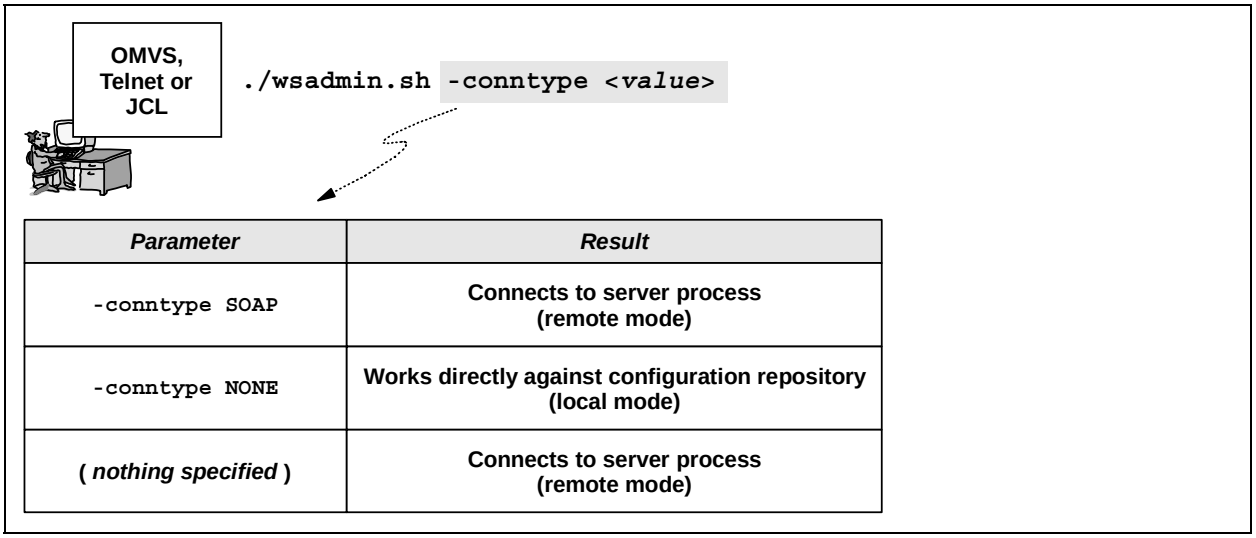
Connecting to server process port, or operating in "local mode"

We've alluded to WSADMIN "connecting" across the network to a TCP port of the running server process. That's known as "remote mode." But there's another way, and it's called "local mode." When operating in "local mode," no TCP connection to a port is attempted. The difference is who (or what) makes the changes to the configuration repository:



Difference between "remote" and "local" mode invocation of WSADMIN

You control which mode WSADMIN will operate in by the parameters you use when invoking the shell script:



Parameter	*Result*
`-conntype SOAP`	**Connects to server process (remote mode)**
`-conntype NONE`	**Works directly against configuration repository (local mode)**
(*nothing specified*)	**Connects to server process (remote mode)**

Controlling the mode in which WSADMIN will operate

Notes:
- If you specify `-conntype SOAP`, two other parameters (`-host` and `-port`) are necessary
- `RMI` is another option for `-conntype`, but we'll ignore that and focus on `SOAP` vs. `NONE` only.

When should one use "local mode" versus "remote mode"?

Some of the WSADMIN functions only work when connected to a real live server process (specifically, the `$AdminControl` object requires a server connection). But the others do not. For those functions that work either way, you could use either method. But there are some general guidelines we would recommend:

- ***If Admin Application server process is <u>not</u> running, then use `-conntype NONE`***

  This makes some sense: if there's nothing running, there's nothing to connect to. Installing applications when the server is not running is a very common thing to do. The `BBOWIAPP` customized job did that when you were configuring WebSphere.

- ***If Admin Application server process <u>is</u> running, connect to it***

  This isn't strictly required, but it's a good practice. If the Deployment Manager is running, connecting to it to drive the changes will help make sure the changes are coordinated. This is particularly true if someone happens to be on the Admin Console at the same time.

  Note: Using WSADMIN concurrent with Admin Console operations is *not* recommended. See "Important: Do NOT use WSADMIN and Admin Console at the same time" on page 12.

  Another benefit of connect to the running process if it's available is that it gives you the full range of WSADMIN functions.

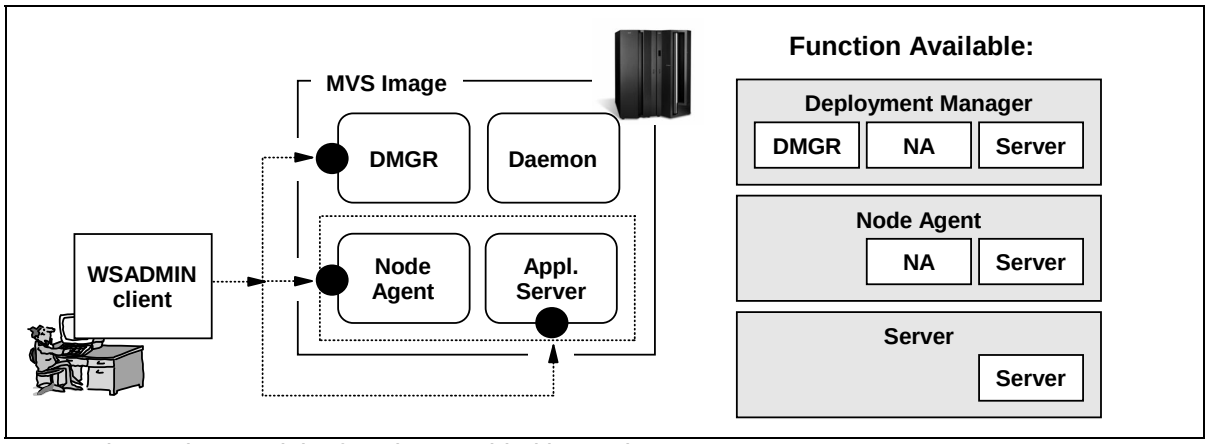
  An exception to this rule is when what you're looking to do with WSADMIN involves no changes to the configuration HFS. Then connecting is not so critical. So for minor things like using the "help" facility to research a command syntax or using WSADMIN to list the applications installed in a server, connecting to the running process isn't really that important.

If connecting, which server process should we connect to?

There are three different server processes you *could* connect to:

- Deployment Manager ⇦ *we're going to recommend connecting to this*
- Node Agents
- Application Servers

The higher up the chain you go, the more function you have available to you. The Deployment Manager is the king of the WebSphere world; it has all the management function available to it. If you connect to a node agent you have a subset of the function; if you connect to an application server you have a still smaller subset.



Connection options and the function provided by each

Note: If you connect to a Node Agent, you may affect servers in that node, but not others nodes. If you connect to an application server, you affect only that server.

We recommend the following:

Network Deployment configuration . . . . . *Connect to Deployment Manager*

Base Application Server node configuration . . . . . *Connect to application server*

> **Note:** In a Network Deployment configuration, the DMGR maintains the "master configuration." It is best to have it make changes to the "master configuration" and then "synchronize" those changes out to the nodes. Changes made at a lower level will be lost the next time the "master configuration" is synchronized out. There's no way to "reverse synchronize." Therefore, it makes sense to use the DMGR to make the changes.
>
> If you think about it, this is exactly what happens when you make changes through the Admin Console. That is merely a GUI interface that operates against the Deployment Manager's "managed process."

We're not sure why one would consider connecting to a Node Agent or an application server when the Deployment Manager is present. If there's a good reason to do that, we've not heard it.

If we operate in local mode (no connection), what config HFS do we act against?

It'll work against the configuration HFS related to the copy of `wsadmin.sh` you invoked. Recall that we said there were multiple copies of that shell script maintained in the configuration HFS. If you invoked `wsadmin.sh` from the Deployment Manager's `/bin` directory, it will operate against the Deployment Manager's "master configuration." If you invoke the copy of `wsadmin.sh` down in an application server's `/bin` directory, it'll operate against that server's configuration.

> **Note:** When using "local mode" in a Network Deployment configuration, it's best to use the copy of `wsadmin.sh` located in the Deployment's `/bin` directory. Make changes against the "master configuration." When node synchronization occurs at some future point in time, those changes will be copied out to the nodes.

Exception: using securityoff command to disable Global Security

The topic of security is way beyond the scope of this document, but there is something that bears mentioning here. The process of enabling Global Security has some risk associated with it: do something wrong, and your servers may not start the next time. Fortunately, there's a WSADMIN command to disable -- or undo -- the Global Security setting within WebSphere.

That command -- `securityoff` -- is used in local mode because local mode will work directly against the configuration HFS when the servers aren't up. If Global Security is misconfigured and the server's *won't* come up, being able to change the configuration in local mode is a very handy thing.

We bring this up because to disable Global Security in a Network Deployment configuration requires that `securityoff` be run against your Deployment Manager *and all the nodes and applications servers in the configuration*. That implies invoking the copy of `wsadmin.sh` in your Deployment Manager's `/bin` directory and invoking the copy in your application servers' `/bin` directory.

For more on enabling and disabling Global Security, see:

`http://www.ibm.com/support/techdocs/atsmastr.nsf/WebIndex/TD101150`

Important: Do NOT use WSADMIN and Admin Console at the same time

The issue here is that you want to avoid having two different processes modifying the configuration at the same time. There's a couple different flavors to this issue:

1. *WSADMIN connected to server process while someone is working at Admin Console*

   This works *most of the time*, but we've seen cases where configuration buffering takes place and a change to the configuration made by WSADMIN doesn't show up on the Admin

Console. The same in reverse: a change made in the Admin Console isn't "seen" by WSADMIN. It can get confusing. The best course of action is to avoid this.

> **Note:** Yes, we recognize this might be hard to avoid. All we ask is that you understand the issue.

2. *WSADMIN in local mode while someone is working at Admin Console*

   This is the worst scenario. Here WSADMIN is making changes *directly* to the configuration HFS. The Admin Console process has *no knowledge* of what's going on until after the fact. It then sees that the configuration has mysteriously changed under its feet. If someone is in the middle of making a change in the Admin Console, that change may have to be discarded. It gets *very* confusing, very quickly.

   > **Note:** This is why we recommend that if the server process that runs the Admin Console is up and running that you connect to it. It's not strictly required, but in the off-chance someone *is* using the Admin Console concurrently, at least things have a reasonable chance of working.

The key message here is this:

WSADMIN or Admin Console, but not both at the same time.

Using a scripting language

WSADMIN is called a "scripting interface" because it provides a way for a scripting language to drive the WSADMIN objects. At the time of the writing of this document the most common scripting language used was "Jacl," a Java version of the older "tcl" language.

> **Note:** WebSphere Application Server for z/OS Version 5.1, when that becomes available, will support the "Jython" scripting language as well. This document uses Jacl examples simply because that's what we know is supported right now, and because most of the examples in the InfoCenter are in Jacl. But more and more Jython examples are showing up in the InfoCenter as well. Be aware of which type you use.

Using a scripting language becomes important because it provides some key things:

- The ability to save the script file and re-use it later
- The ability to execute multiple WSADMIN commands in a row automatically
- The ability to put the value of something in a variable, then use that variable in a WSADMIN command later
- The ability to use scripting language functions such as looping, if-then-else, passed-in parameters, etc.

What you'll see in this document is a mixture of both simple WSADMIN commands and Jacl scripts. As we build upon the lessons and make them more complex, you'll see more and more Jacl scripts.

Learning challenges

The task of learning WSADMIN involves learning two things:

- *The syntax of the WSADMIN commands themselves*
  Every WSADMIN object -- `AdminApp`, `AdminConfig`, `AdminControl` -- has multiple methods, and each method has parameters and options. Becoming skilled in WSADMIN involves understanding those, and understanding where to go to learn more about them.
- *The syntax of the scripting language used*
  The Jacl scripting language has many useful functions -- for instance, `list`, `lappend`, `lindex` -- and syntax requirements involving brackets and braces. The script interpreter can be somewhat finicky about these things.

That is the purpose of this white paper: to help you start the process of learning these things.

Lesson 1: Starting Client and Exploring Commands

Disclaimer: The examples and lessons provided in this document have been tested and are believed to be functional. But things sometimes change. Also, some of the things in these lessons may not be the most efficient or most elegant way to accomplish a task. But they should work. Treat them as what they are: examples used to illustrate key points about WSADMIN.

In this lesson we'll use the scripting interface when it's *not* connected to any server process. This is known as *local mode*. Some functions are not available in this mode (such as the `$AdminControl` functions, which requires that the client be connected to a live server process.) But other functions *are* available, such as the ability to install an application or delete a server.

In a later lesson we'll invoke the client and connect it to a running process and exercise those functions as well.

Access command line interface

The `wsadmin.sh` shell script, like any shell script, may be invoked from a command line interface or in batch mode. For this lesson we'll use the command line interface because it provides real-time interaction. For z/OS, two command line interfaces are available:

- The OMVS shell
- A telnet session

Both accomplish the same thing. For the sake of this document we'll assume the `OMVS` shell.

☐ Log onto TSO and go into the OMVS shell

☐ Use the **`su`** command to switch users to the WebSphere Administrator ID for your cell.

> **Note:** Or you could `su` to a superuser, but that's always a bit risky. The WebSphere Administrator ID will have the authority needed to write to the directories used by the WSADMIN client.

☐ Change directories to the `/<config root>/DeploymentManager/bin` directory

> **Note:** Or the `/<config root>/AppServer/bin` directory if your configuration is a Base Application Server node.

Invoke "Help" facility of WSADMIN

☐ Issue the following command:

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

You should see the following:

```
WASX7001I: wsadmin is the 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]
        [ -conntype
                SOAP
                        [-host host_name]
                        [-port port_number]
                        [-user userid]
                        [-password password] |
                RMI
                        [-host host_name]
                        [-port port_number]
                        [-user userid]
                        [-password password] |
                JMS <jms parms> |
                NONE
        ]
        [ script parameters ]

Where...
```

Note: depending on your display setup, the left square bracket and right square bracket may appear as different characters.

Output from the "help" facility of WSADMIN

Note: The square bracket thing is a long-standing issue with character translation between EBCDIC and ASCII for those characters. If you used Telnet to get to the z/OS system, you'd see the square brackets properly. In any event, the brackets are *not* part of the command, but rather imply the parameter is optional.

Start WSADMIN client interface with -conntype of NONE

At this point the WSADMIN client is not yet started. The previous step simply produced the help output from the shell script itself.

☐ Issue the following command:

```
./wsadmin.sh -conntype none
```

```
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>
```

WSADMIN prompt

Starting the WSADMIN client and gaining access to command prompt

The WSADMIN client is now started. If you had started the client with no parameters at all, it would have by default connected tried to connect to the Deployment Manager process.

??? How would it know what host and port to connect to? The `wsadmin.sh` shell script is located in the `/bin` directory of your Deployment Manager's configuration directory. The shell script then interrogates the XML files in the Deployment Manager's configuration and figures out the host and port value of the Deployment Manager, then connects.

☐ Now issue the **`$Help help`** command to get a listing of the commands available:

```
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
wsadmin                                help text for the wsadmin script

message                                id, returns explanation and
                                       age
```

The three areas where we'll focus in this document

The results from `$Help help` at the wsadmin command prompt

☐ Let's take a look at the help under the `AdminApp` function. Issue the following command:

`$AdminApp help`

```
edit            Edit the properties of an application
editInteractive Edit the properties of
export          Export application to a
exportDDL       Export DDL from applica
help            Show help information

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

installInteractive
                Install                                    ven a

list            List all installed applications
listModules     List the modules in
options         Shows the options a                        in
                general.
publishWSDL     Publish WSDL files for a given application
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
updateAccessIDs Updates the us
                from user regi
deleteUserAndGroupEntries
                Deletes all the user/group information for all
                the roles and all the username/password information for RunAs
                roles for a given application.
```

Method you just invoked to get this screen

Later we'll use this to install an application into a server without using the Admin Console

We'll use this in this lesson to list the applications installed in the cell

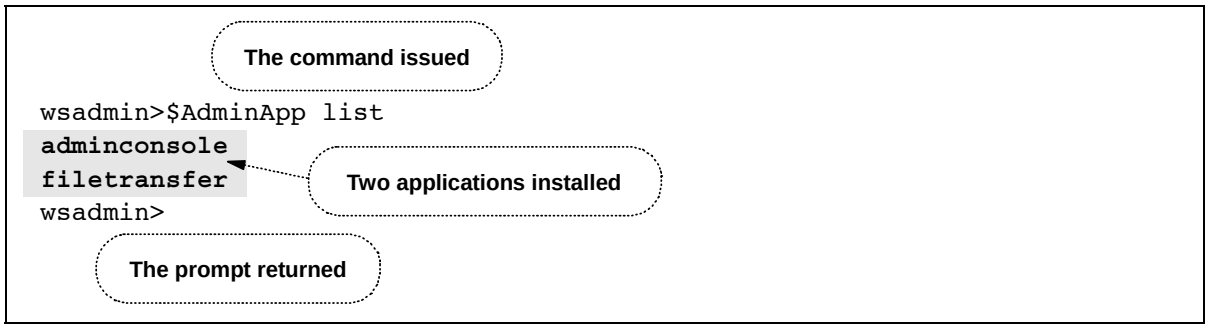
Finally, we'll use the uninstall method to remove an application from the server

Output from the `help` method of the `$AdminApp` function

List the applications installed in this cell

The method `list` on the `$AdminApp` function will display back all the applications that are installed in this cell.

☐ Issue the command **`$AdminApp list`**



Listing the applications installed with the list method

??? If it's not connected to any running server process, how does it know what applications are installed? It looked in the XML files of the configuration.

Exiting the WSADMIN interactive client

☐ Issue the command **`exit`**. You'll be returned to the OMVS prompt.

Concluding points on this lesson

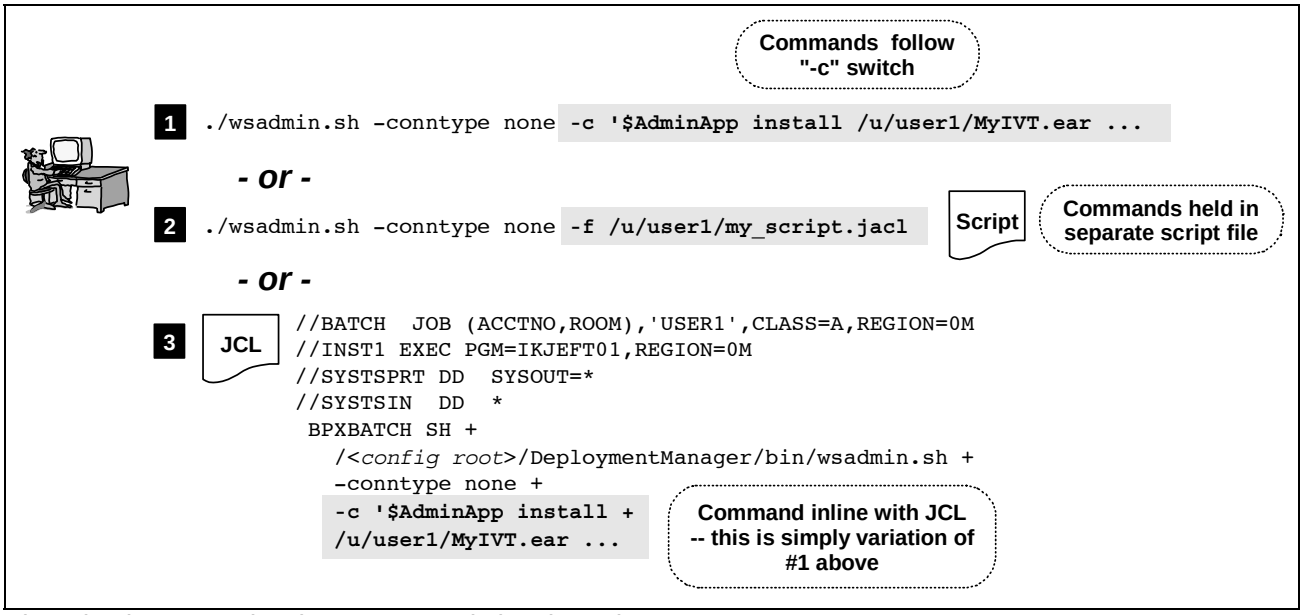
This was a very simple introduction to the WSADMIN function. In this lesson all we did was to invoke a few "help" commands and list the applications. It's helpful to understand what we did not do in this lesson:

- We did *not* connect to a running server process
- We did *not* make any configuration changes
- We made virtually *no* use of Jacl scripting

All that will come later. The importance of this lesson was to understand that the WSADMIN client is the interface into the scripting function of WebSphere. Commands are entered into the client, which executes them on your behalf. So far we've entered the commands interactively, with our own hands. Later we'll put the commands in a file and tell WSADMIN to read and execute the commands in the file. And we'll do that in batch mode.

Lesson 2: Invoking WSADMIN in Batch Mode

The previous lesson focused on issuing WSADMIN commands in interactive mode. WSADMIN is perhaps more useful in batch mode. This can be done from an OMVS (or telnet) prompt, or through JCL. There are three basic ways WSADMIN commands can be "batched up" and fed into the scripting client:



Three basic ways to invoke WSADMIN in batch mode

In all three the processing is essentially the same:

- WSADMIN client starts
- Commands (or script) read in and executed
- WSADMIN client closes

WSADMIN commands that follow the invocation of the shell script

☐ From the OMVS prompt, issue the command:

```
./wsadmin.sh -conntype none -c '$AdminApp help'
```

You should receive the help display for the `$AdminApp` command.

> **Note:** The single quotes surrounding `$AdminApp help` is what allows the command processor to see `help` as a parameter to `$AdminApp`, and not a separate command. If you leave the single-quotes off, WSADMIN will throw an error saying it doesn't recognize `help` as a command. (`$Help` *is* a command, but `help` is not.)
>
> Later we'll see how "braces" -- `{` and `}` -- will be used to do something similar: group parameters and options within a command.

☐ Try another:

```
./wsadmin.sh -conntype none -c '$AdminApp list'
```

You should receive a listing of whatever applications are installed in your cell.

☐ Final one in this section: string two commands together:

```
./wsadmin.sh -conntype none -c '$AdminApp list' -c '$Help help'
```

You'll see the applications installed and then the general help output.

Note: Issuing serial commands like this on the command line does not perform as well as issuing them from inside a file.

WSADMIN commands held in a separate file

In this section we'll make use of the `-f` switch to point WSADMIN off to a separate file.

Important point: WSADMIN by default expects files to be in ASCII

As the heading indicates, by default WSADMIN will expect the file pointed to by the `-f` switch to be in ASCII. By default, files you create in the z/OS HFS will be EBCDIC. If WSADMIN *thinks* its reading ASCII but really is getting an EBCDIC file, things don't work.

At this point you have two options:

1. Create the input file in ASCII on your workstation, then upload it in binary to the z/OS system. This will result in the file being in ASCII in the z/OS HFS.
2. Create the file in EBCDIC. Use the `-javaoption` switch to pass in a parameter that will tell WSADMIN to expect the file to be in EBCDIC.

Aside from the `-javaoption -Dscript.encoding=Cp1037` parameter, there's no difference between the two. We'll focus on the EBCDIC option.

Create HFS file and enter commands

- ☐ Create an HFS file in, for example, `/u/user1` and call it `lesson2a.jacl`

Note: See "Appendix A: Exercises (available for copy-and-paste)" starting on page 94 for a listing of all the lessons provided in this document.

- ☐ Make sure it has permissions that will allow the "WebSphere Administrator ID" access to read the file. (Permissions of `777` will guarantee this.)
- ☐ Add the following commands:

```
EDIT       /u/user1/lesson2a.jacl
Command ===>
****** ****************************
000001 set list [$AdminApp list]
000002 puts stdout $list
****** ****************************
```

Note: the square brackets *must* be the following EBCDIC hex:
[= x'AD'
] = x'BD'

`lesson2a.jacl` - *Simple Jacl script*

Notes:
- Depending on how your emulator is configured, the left and right square bracket characters may not display as brackets. Usually they appear as the following characters:

  Left square bracket: **Ý**

  Right square bracket: **¨**

  The key is they must be hex `x'AD'` and `x'BD'` respectively. If your emulator puts `x'BA'` and `x'BB'` respectively, WSADMIN will fail evaluating the Jacl script.
- If you're having trouble creating the file from your keyboard, create the file on your PC and FTP it to z/OS using ASCII mode. That will perform an ASCII-to-EBCDIC translation and the brackets should appear as `x'AD'` and `x'BD'` respectively.
- What's all that stuff wrapped around the `$AdminApp list` command? That's Jacl, and that's what gets the output back to the screen. We'll go deeper into Jacl programming in "Lesson 3: Introduction to Jacl scripting" on starting on page 25.

- ☐ Save the file.

Intentionally create error where EBCDIC used when WSADMIN expects ASCII

Note: We're doing this simply to show the error you will see.

- ☐ Open an OMVS shell and switch users ("su") to your WebSphere Administrator ID.
- ☐ Change directories to /<*config root*>/DeploymentManager/bin
- ☐ Issue the following command:

```
./wsadmin.sh -conntype none -f /u/user1/lesson2a.jacl
```

What was your result? You likely saw something like this:

```
WASX7017E: Exception received while running file "/u/user1/lesson2a.jacl";
exception information: com.ibm.bsf.BSFException: error while
eval'ing Jacl expression: missing close-bracket
```

Error message received at this point in the lesson

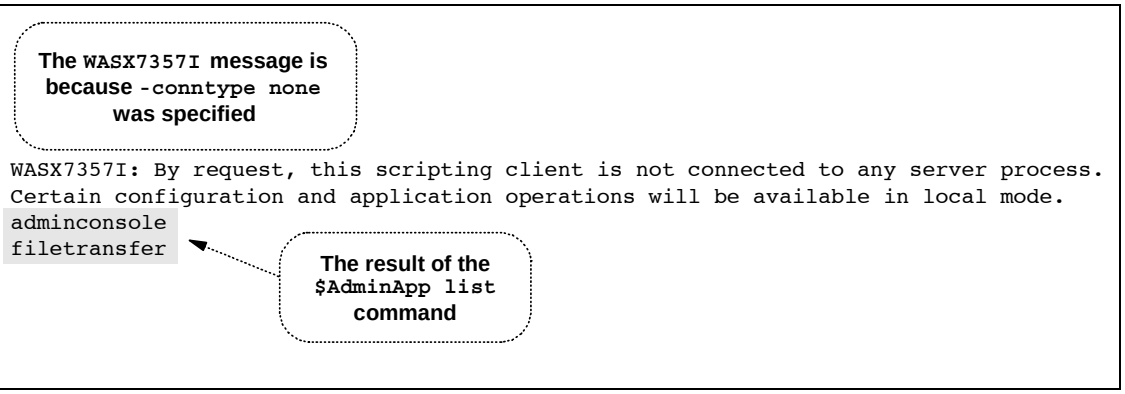
Note: The problem is *not* a missing close-bracket. The problem is WSADMIN was expecting certain things, and since the file was in EBCDIC, it didn't see them. We'll fix that next.

Use -javaoption switch to indicate source is in EBCDIC

- ☐ Issue the following command *as one input string*:

```
./wsadmin.sh -javaoption -Dscript.encoding=Cp1047
                                    -conntype none -f /u/user1/lesson2a.jacl
```

Now you should see this:



Successful execution of script in external file

Note: That might seem like a lot of keystrokes to achieve the same result you would get if you simply typed '$AdminApp list' on the command line. That's true ... it was. Bear in mind that the sample Jacl script we provided was incredibly simple. Normally Jacl scripts are longer, more complicated, and do many more things. In that case, entering the command above is a small price to pay for the processing that occurs when the script is executed.

WSADMIN commands inside JCL

Having the command inside JCL is, as mentioned earlier, simply a variation on entering the command after the -c switch on the wsadmin.sh invocation. The difference is, of course, that

once you have it coded and working, you can submit it over and over again with a simple SUB command.

A look inside the BBODIAPP job

To illustrate how this works, we'll explore the BBODIAPP job that was created when you customized your Deployment Manager. (Or BBOWIAPP if what you have is a Base Application Server node.) That job installs the administrative application into the server. That job is nothing more than BPXBATCH running of wsadmin.sh, with a long string of command input. The $AdminApp install command is executed with a whole bunch of options. Here's what the job looks like:

```
//SYSTSIN  DD  *
BPXBATCH SH +
  /wasv5config/azcell+
  /DeploymentManager+
  /bin/wsadmin.sh -conntype none +
  -c '$AdminApp install +
  /wasv5config/azcell+
  /DeploymentManager+
  /installableApps/adminconsole.ear +
   {-appname adminconsole  +
   -MapRolesToUsers {{"administrator" No No AZADMIN AZCFG} +
   {"monitor" No No AZADMIN AZCFG} +
   {"operator" No No AZADMIN AZCFG} +
   {"configurator" No No AZADMIN AZCFG}} +
  -server dmgr +
  -node azdm  +
  -cell azcell +
  -copy.sessionmgr.servername +
  dmgr}' +
  1> /tmp/bbodiapp_40237.out +
  2> /tmp/bbodiapp_40237.err
/*
```

A peek inside the BBODIAPP batch job that installs the Admin Console into the DMGR

1. The JCL simply invokes BPXBATCH SH, which is used to run the wsadmin.sh shell script
2. A plus sign ("+") is used to continue lines when the command is in JCL. But this is not the case for commands inside Jacl scripts. We cover that later.
3. The wsadmin.sh shell script is located in the /DeploymentManager/bin directory under the configuration root for this cell.
4. The flag -conntype none is used for a very good reason: the Deployment Manager is not yet up, so there's nothing to connect to. This illustrates an important point: the DMGR does not need to be running to make updates to the configuration. The configuration is simple XML files and directories. The shell script can make changes to those even though no server process is running.
5. The WSADMIN command used in this job was $AdminApp install
6. The application being installed was adminconsole.ear, located in the directory shown above
7. The things that follow the EAR file are the options to the install method of $AdminApp. This is "graduate level" stuff. We'll cover *some* of it later under "Lesson 4: Installing an Application using $AdminApp Object" on page 35 when we walk you through installing of an application. We'll cover more under "Lesson 7: Digging Deeper into the $AdminApp Object" on page 80.
8. Finally, the stdout ("1") and stderr ("2") of this shell script is piped to a file in the /tmp directory. The JCL later copies that file back to the job log.

Note: If you look at the JCL for `BBODIAPP`, you see that it actually has *three* steps in it: the first installs `adminconsole.ear` (shown above); the second installs `filetransfer.ear`; and the final step copies the `stdout` and `stderr` back to the job log.

Question: why no -javaoption in JCL to say commands are in EBCDIC?

Because the commands are their parameters are not in a separate script file. They're inline, following the invocation of the `wsadmin.sh` shell script. The script interpreter assumes what follows the invocation of the shell script is in the same code page as the environment.

It's only when an external file is used to contain the commands that we need to tell the interpreter that the code page of the file is `Cp1037` (or EBCDIC).

Copy BBODIAPP job and modify it to do something simpler at this point

- ☐ Make a copy of `BBODIAPP` and call it something else ... `LESSON2B` for example.
- ☐ Remove the `INST2 EXEC` section ... that's just the same thing as `INST1` but with a different application. Leave the `DIAPPC EXEC` section.
- ☐ Modify the body of the JCL so it looks like this:

```
//*********************************************
//*  STEP 1 - Invoke WSADMIN, issue command
//*********************************************
//INST1 EXEC PGM=IKJEFT01,REGION=0M
//SYSTSPRT DD  SYSOUT=*
//SYSTSIN  DD  *
 BPXBATCH SH +
   /wasv5config/azcell+
   /DeploymentManager+
   /bin/wsadmin.sh -conntype none +
   -c '$AdminApp list' +
   1> /tmp/lesson2b.out +
   2> /tmp/lesson2b.err
/*
//************************************
//*  STEP Copy - Copy script output back to joblog
//*********************************************
//DIAPPC EXEC PGM=IKJEFT01,REGION=0M
//SYSEXEC  DD  DISP=SHR,DSN=WAS502.WAS.SBBOEXEC
//SYSTSIN  DD *
   BBOHFSWR '/tmp/lesson2b.out'
   BBOHFSWR '/tmp/lesson2b.err'
//SYSTSPRT DD SYSOUT=*
//
```

Point to your config root 1

Change command to $AdminApp list 2

Pipe STDOUT and STDERR to new file, and make sure that those files are copied back to joblog 3

LESSON2B - *Modified BBODIAPP used to execute `$AdminApp list` command*

1. Be careful with the plus sign continuation ... that should *immediately follow* (no space) your configuration root value. The next line, `/DeploymentManager`, should concatenate directly onto your config root.
2. Be sure to enclose the command and its parameter with single quotes, as shown. And in this case the plus sign follows the command with a space. We do *not* want the next line to concatenate directly onto this. We want a space between the end of the command and the `1>` piping of the `STDOUT`.
3. The name of the file can be anything you like. The key is being consistent between where you pipe it and what file you copy back to the job log. Also, piping the output to /tmp is a good idea because the permissions on that directory should allow this process to write to it without difficulty.

- ☐ Provide a JOB card. Make sure job runs under identify of "WebSphere Administrator ID" (or ID connected to the "WebSphere Configuration Group").
- ☐ Submit the job. You should get a response back that looks something like this:

```
 :
READY
   BBOHFSWR '/tmp/lesson2.out'
WASX7357I: By request, this scripting client is not connected to any
server process. Certain configuration and application operations will
be available in local mode.
adminconsole
filetransfer
READY
   BBOHFSWR '/tmp/lesson2b.err'
READY
END
```

Result we were looking for

Output from JCL batch submission of `$AdminApp list` command

Pointing to an external file from a JCL batch job

This is a variation of the two previous lessons: an external file which holds the commands, and a JCL batch job that invokes WSADMIN and then points to the external file.

- ☐ Make a copy of the file `/u/user1/lesson2a.jacl` and call the new file `/u/user1/lesson2c.jacl`

> **Note:** Don't change the contents of the file. We'll run the exact same commands. We'll just invoke WSADMIN through batch JCL rather than from the OMVS command prompt.

- ☐ Make a copy of your `LESSON2B` member and call it `LESSON2C`
- ☐ Modify the body of the JCL so it looks like this:

```
//*********************************************
//*  STEP 1 - Invoke WSADMIN, point to file
//*********************************************
//INST1 EXEC PGM=IKJEFT01,REGION=0M
//SYSTSPRT DD  SYSOUT=*
//SYSTSIN  DD  *
 BPXBATCH SH +
   /wasv5config/azcell+
   /DeploymentManager+
   /bin/wsadmin.sh +
   -javaoption -Dscript.encoding=Cp1047 +
   -conntype none +
   -f /u/user1/lesson2c.jacl +
   1> /tmp/lesson2c.out +
   2> /tmp/lesson2c.err
/*
//*********************************************
//*  STEP Copy - Copy script output back to joblog
//*********************************************
//DIAPPC EXEC PGM=IKJEFT01,REGION=0M
//SYSEXEC  DD  DISP=SHR,DSN=WAS502.WAS.SBBOEXEC
//SYSTSIN  DD *
   BBOHFSWR '/tmp/lesson2c.out'
   BBOHFSWR '/tmp/lesson2c.err'
//SYSTSPRT DD SYSOUT=*
//
```

Provide `-javaoption` to tell interpreter script file is in EBCDIC

Provide `-f` switch and pointer to location of external file

`LESSON2C` - *Batch JCL that points to external script file which holds WSADMIN commands*

Concluding points on this lesson

If WSADMIN was a command-line interface only and required the hand-input of long strings of commands each time, it would quickly become drudgery. Thankfully the process can be done in batch, which allows you to save and re-use things you've already developed.

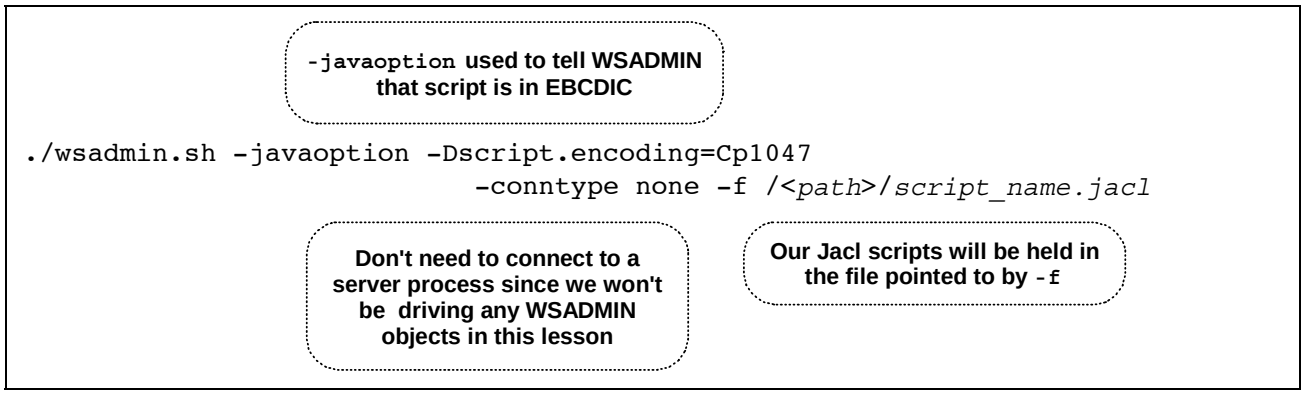
The stage is now set for an introduction to Jacl scripting, and then onto the more complex WSADMIN operations, such as controlling the runtime and installing applications.

Note: The majority of examples in this document will show the Jacl script being invoked from an OMVS (or Telnet) session. But JCL could be used just as easily. Remember that as you read through this document: *you may use JCL to invoke Jacl scripts*.

Lesson 3: Introduction to Jacl scripting

In the previous lesson we illustrated ways to pass in strings of input into WSADMIN. In all cases what was being passed in was Jacl script, though very simple examples of it. Now we're going to explore the Jacl scripting language a bit more.

In this lesson we'll explore some basic aspects of the Jacl language. *We won't involve any WSADMIN commands in this lesson*. We'll start folding WSADMIN stuff into more complex Jacl in the next lesson. The method we'll use to invoke the Jacl script will be what we demonstrated in the previous lesson:



Use `-f` switch of `wsadmin.sh` to point to Jacl script to be evaluated and executed

Note: If you prefer to create your script on the PC and upload in binary so it's in ASCII on the host, then you do not need to code the `-Dscript.encoding=` option.

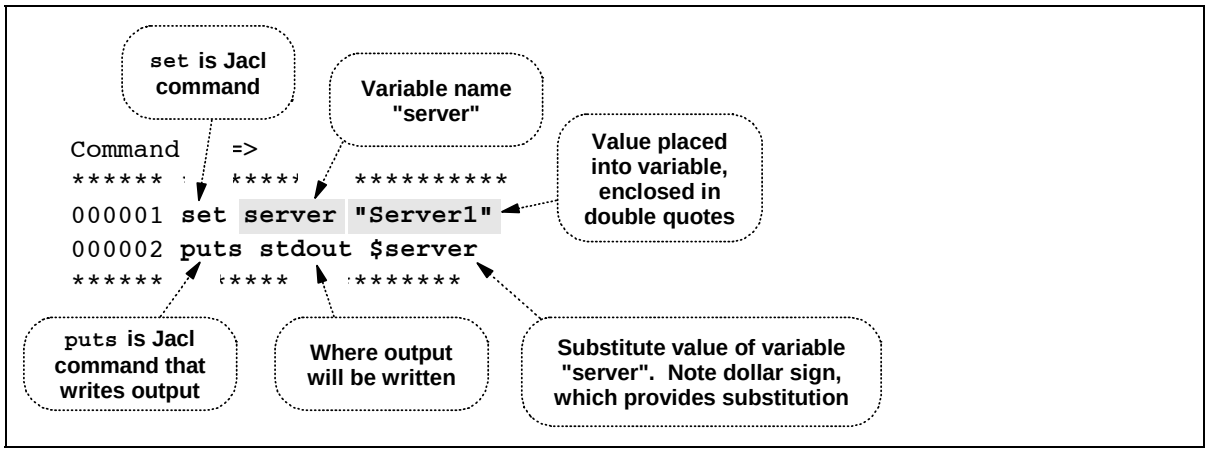
Getting ready

- ☐ If you're not already there, go to an `OMVS` shell and `su` to your WebSphere Administrator ID. Go to the `/<config root>/DeploymentManager/bin` directory.

  Notes:
  - Or the `/config root>/AppServer/bin` directory if Base Application Server node.
  - This is a good point to say, "Or, if you prefer, open a telnet session." Both achieve the same result: access to the z/OS Unix Systems Services command prompt.

Simple setting of variable and putting variable back to screen

- ☐ Create a file called `lesson3a.jacl` in your home directory. (For the sake of this document, we'll assume that directory is `/u/user1`.)
- ☐ Edit the file and add the following:



lesson3a.jacl - *Single variable set in Jacl script*

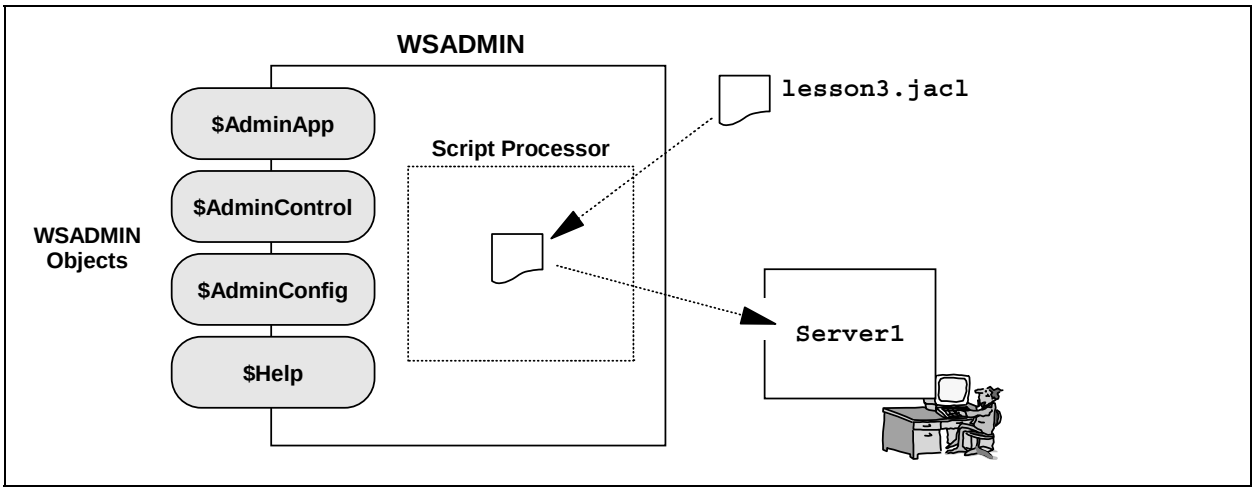
☐ Execute the script with the following command (*all on one line*):

```
./wsadmin.sh -javaoption -Dscript.encoding=Cp1047 -conntype none
                                                  -f /u/user1/lesson3a.jacl
```

You should receive `Server1` back to your screen.

What happened in that last exercise?

Not much. The script was read into WSADMIN, evaluated, and the output was written back to the terminal. None of the WSADMIN objects were used:



Script did not make use of any of the WSADMIN objects

Note: But we'll change that in the next lesson. It's no coincidence that the variable we set was "server."

Setting multiple variables

☐ Now create a file called `lesson3b.jacl` and set multiple variables:

```
set cell    "mycell"
set node    "mynode"
set server  "Server1"
set appl    "my_appl"
puts stdout "C:$cell N:$node S:$server A:$appl"
```

lesson3b.jacl - *Multiple variables set in script, displayed back*

☐ Invoke `wsadmin.sh` with the following command:

```
./wsadmin.sh -javaoption -Dscript.encoding=Cp1047 -conntype none
                                                  -f /u/user1/lesson3b.jacl
```

You should receive the following response:

```
C:mycell N:mynode S:Server1 A:my_appl
```

Parsing words out of a string

☐ Create `lesson3c.jacl` and provide the following:

```
set string  "Four Score and Seven"
set first   [lindex $string 0]
set second  [lindex $string 1]
set third   [lindex $string 2]
set fourth  [lindex $string 3]
puts stdout "1st:$first 2nd:$second 3rd:$third 4th:$fourth"
```

`lesson3c.jacl` - *Using Jacl `lindex` function to parse out words from a string*

Important: For this to work, square brackets must be `x'AD'` and `x'BD'` in the file on z/OS.

??? `lindex` is a Jacl function that will parse values of a string, starting with a zero offset.

☐ Issue the command

```
./wsadmin.sh -javaoption -Dscript.encoding=Cp1047 -conntype none
                                                  -f /u/user1/lesson3c.jacl
```

You should see:

```
1st:Four 2nd:Score 3rd:and 4th:Seven
```

??? Pure gee-whiz? Not at all. Please continue.

Passing values in as parameters

Rather than hard-coding the variable values in the script itself, we'll pass them as parameters on the invocation of the script itself.

☐ Now create a file called `lesson3d.jacl` and provide the following:

```
set cell    [lindex $argv 0]
set node    [lindex $argv 1]
set server  [lindex $argv 2]
set appl    [lindex $argv 3]
puts stdout "C:$cell N:$node S:$server A:$appl"
```

`lesson3d.jacl` - *Using Jacl `lindex` against special variable `$argv`*

Important: For this to work, square brackets must be `x'AD'` and `x'BD'` in the file on z/OS.

??? Here again we use the `lindex` function. `$argv` is a special Jacl variable that represents the string of parameters passed into the script.

☐ Now invoke this with the following command (*all on one line*):

Beginning part of command is the same as before

```
./wsadmin.sh -javaoption -Dscript.encoding=Cp1047 -conntype none

                              -f /u/user1/lesson3d.jacl mycell mynode Server1 my_appl
```

Parameter string passed into the script

Invocation of Jacl script and passing in of parameters

You should see a response back of:

```
C:mycell N:mynode S:Server1 A:my_appl
```

Note: Though that script worked, it has limitations:

- There was no check to insure the minimum number of parameters was supplied, or that the number of parameters supplied was more than the number of variables in the script.
- If a user entered the parameter string in the wrong sequence, the script would simply assign the values to the variables in whatever order they were received

We'll address the first issue next. The second issue is more difficult: it requires validation of the input string against rules designed around a naming convention. That's possible in Jacl, but beyond the scope of this document.

Using If-Else to validate the number of parameters passed in

Suppose your script requires that exactly three parameters be passed in. Here's how you can check that three -- and only three -- were indeed passed in:

☐ Create a file called `lesson3e.jacl` and provide the following:

Important!

Square brackets in EBCDIC must be: `[ = x'AD'` `] = x'BD'`

```
if { !($argc==3) } then {
  puts stdout "You supplied $argc parameters, not three.  Try again"
} else  {
 set cell   [lindex $argv 0]
 set node   [lindex $argv 1]
 set server [lindex $argv 2]
 puts stdout "Parameters: cell:$cell node:$node server:$server"
}
```

`lesson3e.jacl` - *If-Else used to validate exactly three parameters passed in*

There's a lot going on there. Let's explore that in some detail:

```
                  1
2 00001 if { !($argc==3) } then {
                                        3
  00002   puts stdout "You supplied $argc parameters, not three.  Try again"
  00003 } else  {  4
  00004  set cell   [lindex $argv 0]
  00005  set node   [lindex $argv 1]
  00006  set server [lindex $argv 2]
  00007  puts stdout "Parameters: cell:$cell node:$node server:$server"
  00008 }  5
```

If-Else structure explored in more detail

1 The variable `$argc` is a special one that supplies the number of elements in the parameter string.

2 The "if" condition starts. The evaluation is made against the condition `!($argc==3)`, which is "variable `$argc` *not* equal to 3." (The `!` out front provides the "not" aspect of that.) Notice how the whole condition being evaluated is enclosed in *braces*.

3 The "then" clause starts. Note how the open-brace for the "then" clause starts *on the same line* as the close-brace for the evaluation. This is a requirement. One or more spaces must separate the close-brace from the open-brace. The "then" clause is simple: it puts back to the screen a message indicating the number of parameters entered, and how that number is not three.

4 The "else" clause begins. The keyword `else` must follow the close-brace of the if condition; it must be on the same line, and the open-brace must follow that. This "else" clause simply uses the `lindex` function to parse out the parameters and assign them to variables.

5 If the number of parameters passed in was not 3, then processing skips over the "else" clause and the script exits. Had we coded the setting of the variables outside the "else" clause, that stuff would have been processed even if the number of parameters wasn't three. The "if" evaluation of "not equal to three" would have passed, the "then" processing would have occurred, and after that the processing would pick up with what follows. But we coded the parameter processing inside the "else" clause. Therefore, when `!($argc==3)` was met, the message was posted and the script ended.

It *is* possible to force an exit from a script at any point by using the `exit` function. We could have coded `exit` right after the error message and quit the script right there. If that was the case, then the parameter processing stuff could have been outside the "if-then" structure.

Note: As you can see, Jacl is somewhat sensitive about where things appear relative to others. For example, the open-braces had to follow the preceding condition and could not be on the next line. Be careful with that sort of thing.

☐ Invoke the script with the following (*all on one line*):

```
./wsadmin.sh -javaoption -Dscript.encoding=Cp1047 -conntype none

                              -f /u/user1/lesson3e.jacl mycell mynode Server1
```

Parameter string with exactly three arguments

Invocation of Jacl script and passing in of parameters

You should see a response back of:

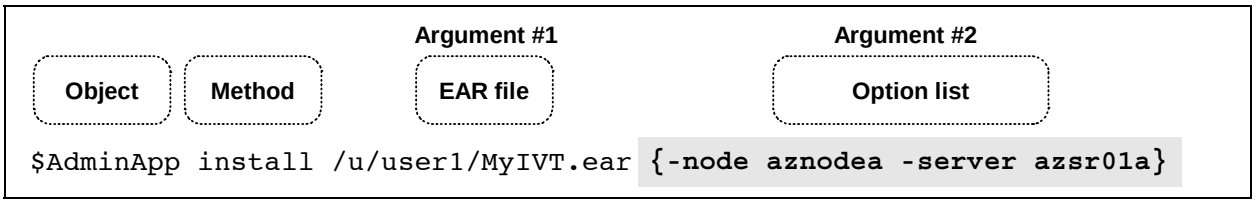
```
cell:mycell node:mynode server:Server1
```

- ☐ Invoke the script again, this time with four parameters. You should see a response back of:

```
You supplied 4 parameters, not three.  Try again
```

Jacl lists -- an important way of providing options to WSADMIN commands

Many of the WSADMIN commands have "options" as one of the arguments (or parameters) to the task being invoked. Those options are coded inside braces -- " { " and " } " -- to gather up the options into a single parameter, even though lots of things may be coded inside the braces. For example, the `install` task of `$AdminApp` object has two arguments: the EAR file being installed and the options to be used. In a Network Deployment configuration the options must contain at least `-node` and `-server` to indicate where the application is to be installed. The syntax of the command would look something like this:



Options are second argument to the install method; are enclosed in "braces"

If you failed to code the braces on that command, WSADMIN would interpret `-node` as the second argument, `aznodea` as the third, `-server` as the fourth and `azsr01a` as the fifth argument. The problem with that is that the install method only has two arguments. WSADMIN would throw an error:

```
... can't find method "install" with 5 argument(s)

                    Arguments:  1            2     3      4      5
  $AdminApp install /u/user1/MyIVT.ear -node aznodea -server azsr01a
```

Error message thrown by WSADMIN when braces not coded to enclose options list

Note: You see the importance of the braces. Proper coding of the braces will become a point of some frustration as you learn this. You'll see later that some options have sub-options, requiring braces inside of braces, all properly matched up and balanced.

Using Jacl variables to break up long command lines

These option lists can get quite long, depending on the number of options being coded onto the task. The previous example showed only two, but if you look at the `BBODIAPP` job (or `BBOWIAPP` if you have a Base Application Server node), you'll see quite a few options. That gets awkward coding all that one command line entry. And if you're coding the command in a separate file ("WSADMIN commands held in a separate file" on page 19) you *still* have to code it on one line.

??? Line continuation inside a Jacl script is tricky. The backslash character (" \ ") works in some cases and not others, depending on where you try to break the line and continue. None of the WSADMIN sample scripts we've seen show command strings broken up with line continuation characters. They use variable substitution instead.

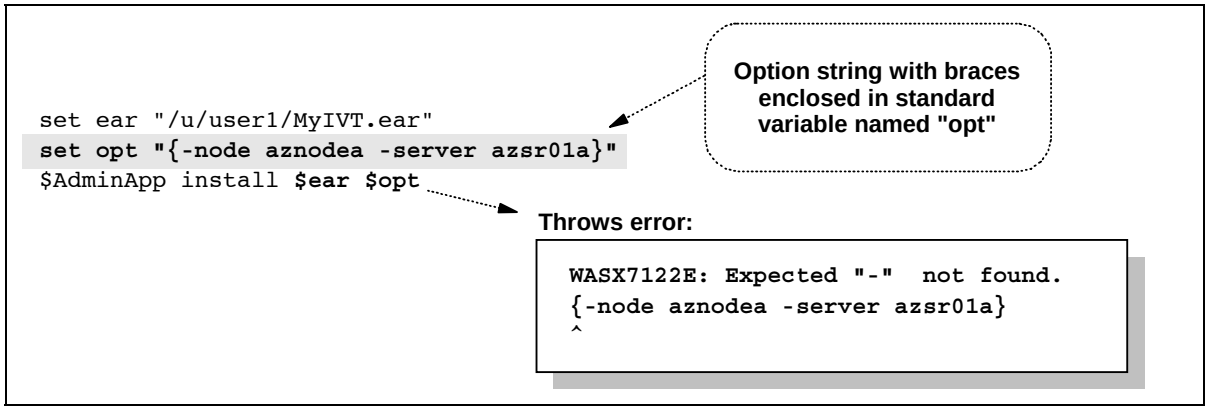
An example of Jacl variables being used to break up a line might look something like this:

```
set ear "/u/user1/MyIVT.ear"

$AdminApp install $ear {-node aznodea -server azsr01a}
```

Jacl script where command line contains variable substitution

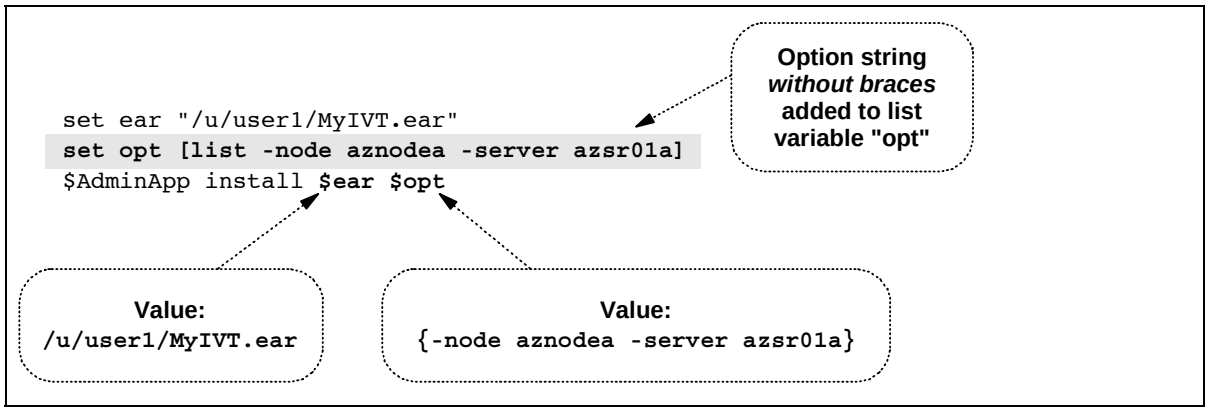
Could the options also be done in the same way? Unfortunately not:



Can't enclose braces in a string variable -- throws error

Jacl "list" function to the rescue

The Jacl language provides a function called `list` that allows a set of arguments or parameters to be held in a variable and passed into a WSADMIN command. To fix the problem in the previous picture all we need do is use the `list` function:



Jacl "list" function solves the option string issue

Note: Notice how the list function is enclosed in square brackets.

So the `list` function provides a set of implied braces around the option list and allows the install method to accept the options as a single argument.

Note: The Jacl `list` function is a widely used and critically important component for WSADMIN coding. It's important to understand how this thing works.

Variable substitution into "list" function

Now we'll do a few more exercises.

☐ Create a file called `lesson3f.jacl` and provide the following:

```
set node "aznodea"
set server "azsr01a"
set ear "/u/user1/MyIVT.ear"
set opt [list -node $node -server $server]
puts stdout "AdminApp install $ear $opt"
```

`lesson3f.jacl` - *Using Jacl `list` function to put argument list into variable*

Notes:
- Square brackets must be `x'AD'` and `x'BD'` EBCDIC for this to work.
- Do *not* code a dollar sign on the front of "AdminApp." Jacl will think that's a variable and try to substitute something into the variable. For now we're just playing around with Jacl and not trying to actually invoke `$AdminApp`.

☐ Invoke that with the following command (*all on one line*):

```
./wsadmin.sh -javaoption -Dscript.encoding=Cp1047 -conntype none
                                                  -f /u/user1/lesson3f.jacl
```

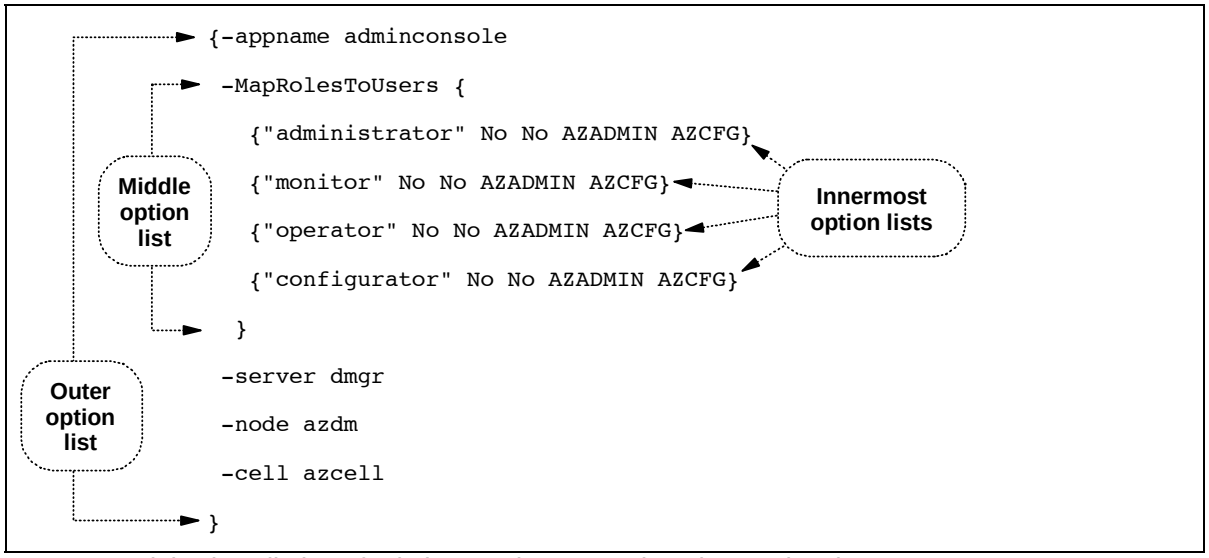
You should receive back:

```
AdminApp install /u/user1/MyIVT.ear -node aznodea -server azsr01a
```

Note: Don't let the absence of braces around the option list concern you. If we were actually invoking `$AdminApp install` the `$opt` variable would imply them ... because we used `[list]` to create the variable.

Nested options -- option lists inside and option list

Back in "A look inside the BBODIAPP job" on page 21 we saw the WSADMIN command that installed the Admin Console application into the Deployment Manager. If you look closely at the option list that followed the `$AdminApp install` command, you'd see there was three levels of option lists at work:



BBODIAPP job's installation of adminconsole.ear employed nested options

Please read: Relax. The `-MapRolesToUsers` task is a complex one and *way* beyond what we're trying to get across here. Don't worry if you don't yet understand that task or its options. That's not the point of this exercise. We're simply using this as an example of nested options in a "real world" setting.

Simplified, we see the following structure:

```
$AdminApp install <ear> {-outer {-middle {inner1} {inner2}}}
```

Even that's a bit complex for us at the moment. Let's focus on a simple nested option:

```
task {-opt1 {opt1a opt1b opt1c}}
```

Let's construct that using the `[list]` function and variable substitution:

☐ Create a file called `lesson3g.jacl` and provide the following:

```
set inner [list opt1a opt1b opt1c]
set outer [list -opt1 $inner]
puts stdout "task $outer"
```

`lesson3g.jacl` - *Putting a "list" inside another "list"*

Notes:

- Square brackets must be `x'AD'` and `x'BD'` EBCDIC for this to work.
- The variable "inner" is being populated with a list of three elements. Braces will be put around these three because the `[list]` function is being used.
- The variable "outer" is being populated with `-opt1` and then the contents of the "inner" list variable.
- Some examples show the list function being nested inside another list function:

  ```
  set outer [list -opt1 [list opt1a opt1b opt1c]]
  ```

  This reduces the number of lines in your Jacl script. But it can tend to be a bit confusing. Throughout this document we'll avoid this. Rather, we'll construct separate variables and build our argument lists step-by-step.

☐ Invoke that script with the following command:

```
./wsadmin.sh -javaoption -Dscript.encoding=Cp1047 -conntype none
                                                  -f /u/user1/lesson3g.jacl
```

You should get a response back of this:

```
task -opt1 {opt1a opt1b opt1c}
```

Outer-most braces not shown in "puts" command, but they're implied and this will work with WSADMIN

Result: option list nested inside another option list

Let's say your objective was something like this:

```
task {-opt1 {opt1a opt1b} -opt2 {opt2a opt2b}}
```

☐ Create a file called `lesson3h.jacl` and provide the following:

```
set inopt1 [list opt1a opt1b]
set inopt2 [list opt2a opt2b]
set outer [list -opt1 $inopt1 -opt2 $inopt2]
puts stdout "task $outer"
```

`lesson3h.jacl` - *Multiple nested lists*

- ☐ Invoke that script with the following command:

```
./wsadmin.sh -javaoption -Dscript.encoding=Cp1047 -conntype none
                                                  -f /u/user1/lesson3h.jacl
```

You should get a response back of this:

```
task –opt1 {opt1a opt1b} –opt2 {opt2a opt2b}
```

Outer-most braces not shown in "puts" command, but they're implied and this will work with WSADMIN

Result: two option lists nested inside another outer option list

Concluding points on this lesson

We now have some of the basics of Jacl in place. Please understand there's a considerable amount of functionality in the Jacl scripting language, and what we've shown you here is but a small fraction of it. But we have some of the key things that will now allow us to fold the WSADMIN objects into the Jacl and start to affect the state of the WebSphere for z/OS runtime.

Lesson 4: Installing an Application using $AdminApp Object

We're now ready to explore using the `$AdminApp` object to install an application into a server. We'll start with a few relatively simple examples, then build on that.

For this exercise we're supply a packaged EAR file called `MyIVT.ear`. That EAR file is relatively simple to install because all its JNDI naming and mapping is done, and there's no data resources to connect to.

Note: What that really means is there isn't a lot of WSADMIN task options to specify to get this application installed. Therefore, it's a great application to use to illustrate the `$AdminApp install` option. You'll see the fundamental things first, then we'll expand on it later.

To connect to a server process or not ... that is the question

Up to this point we've specified `-conntype none` on the `wsadmin.sh` invocation. We did that because it kept things simple. Now we're getting ready to install an application using the `$AdminApp install` function.

Is it now *required* to connect to a server process to make this work? No. If you look at the `BBODIAPP` job (see "A look inside the BBODIAPP job" on page 21) you'll see that the job uses `$AdminApp install` with a `-conntype of none`. But don't think it doesn't make any difference, because it does. Here's the simple rule of thumb:

If the Deployment Manager is *running*	use `-conntype SOAP -host <host> -port <port>`
If the Deployment Manager is *stopped*	use `-conntype none`

Notes:
- If your cell is a Base Application Server node, then substitute word "server" for "Deployment Manager." Or think of it this way: "If the server in which the Admin Application is installed..."
- You may omit the `-conntype`, `-host` and `-port` parameters altogether. It'll default to a protocol of `SOAP`, and it'll connect to the SOAP port of the Deployment Manager defined in the directory structure from which you invoked the `wsadmin.sh` shell script.

This was discussed back under 'When should one use "local mode" versus "remote mode"?' on page 10.

Which server process to connect to?

The `$AdminApp install` function is only applicable to that server process in which the Admin Application is running:

Network Deployment configuration . . . . . Deployment Manager

Base Application Server node configuration . . . . . Base Application Server

If you connect to any other server process, you'll receive this message:

```
WASX7206W: The application management service is not running.
Application management commands will not run.
```

Caution: does *not* mean that server is not running ... means that server is not enabled to support the "Application management" commands.

Error received when attempting $AdminApp install against improper server

If -conntype none used, does it matter which copy of wsadmin.sh used?

There is a copy of `wsadmin.sh` located in the `/bin` directory of the DeploymentManager and the `/bin` directory of each application server node in your configuration. If you plan on using the `-conntype none` option, it *does* matter which copy of `wsadmin.sh` you invoke. This is because the copy of `wsadmin.sh` invoked determines what management services are available, even when `-conntype none` is specified.

In a Network Deployment configuration, if you invoke `wsadmin.sh` from the `/AppServer/bin` directory with `-conntype none` specified and you attempt to invoke `$AdminApp install`, you'll see:

```
WASX7206W: The application management service is not running.
Application management commands will not run.
```

This is because the `/AppServer` directory is an application server node, and in a Network Deployment configuration only the Deployment Manager copy of `wsadmin.sh` has the ability to accept the `$AdminApp install` command.

Notes:
- We're talking about the `$AdminApp install` command only at this point. Other commands may be run against the Node Agent and the application server. But the `$AdminApp install` command is intended to be executed against the server process where the Admin Application is installed. For a Network Deployment configuration, that's the Deployment Manager.
- If your configuration is a Base Application Server node, then the Admin Application will be installed in the application server, and the only copy of `wsadmin.sh` will be under the `/AppServer` directory. `$AdminApp install` *will* work. WSADMIN will understand that the server is not part of a Network Deployment configuration.

Important note concerning server security if enabled

Up to this point we've not worried about the question of authenticating the user invoking the WSADMIN process. That's because we've used `-conntype none` for all the exercises to this point. That option does not require authentication.

Now we're going to start using `-conntype SOAP`, and that option permits a userid and password to be passed in. *But that's only necessary if you have global security enabled for the server*.

The exercises in this document were prepared as if global security was not enabled. Therefore, none of the examples here have the `-user` and `-password` parameter specified. But if global security is enabled on your system, then simply do the following:

If no security enabled:

```
./wsadmin.sh -conntype SOAP -host <host> -port <port>
```

If security enabled:

```
./wsadmin.sh -conntype SOAP -host <host> -port <port> -user <userid> -password <password>
```

Supplying `-user` and `-password` when global security is enabled

Simple install with minimum options

Preliminary activities to ready your environment for exercise

First we'll take care of a few items of business.

Determine the SOAP port of your Deployment Manager (or BaseApp server)

As we mentioned earlier, you *could* simply invoke the `wsadmin.sh` shell script with no connection parameters. That'll default to a SOAP connection and it'll rummage around in the XML file and find the host and port to connect to. For this exercise we'll show an explicit connection.

- ☐ From the Admin Console, drill down and obtain the actual SOAP port being used:

Admin Console Navigation
System Administration
Deployment Manager
End Points
SOAP Connector Address

Location in the Admin Console navigation where SOAP port is displayed

Note: For a Base Application Server node, that would be: *Servers, Application Server, <server>, End Points, SOAP Connector Address.*

Make sure Deployment Manager (or BaseApp server) is up and running

- ☐ To connect to the server process it must be up and bound to the TCP port. If you can log onto the Admin Console, it's up.

Open OMVS shell (or telnet)

- ☐ Use the `su` command to switch to the "WebSphere Administrator ID."

Note: The purpose for doing this is so you'll have the authority to write files into various directories. This is *not* the same thing as the issue of "global security" covered in "Important note concerning server security if enabled" on page 36.

- ☐ Navigate to the `/<config root>/DeploymentManager/bin` directory.

Note: Or `/<config root>/AppServer/bin` if Base Application Server node

Place MyIVT.ear in the HFS

- ☐ FTP the `MyIVT.ear` file in binary mode and place it in the `/u/user1` directory

Invoke "help" to understand the $AdminApp object better

- ☐ At the OMVS command prompt, issue the command:

```
./wsadmin.sh -conntype none
```

This will take invoke WSADMIN and produce a WSADMIN prompt.

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

```
$AdminApp help
```

You should get a listing that looks like this:

Methods on the `$AdminApp` object

```
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
help            Show help information
install         Installs an application, given a file name and an option string.
installInteractive
                Installs an application in interactive mod
                file name and an option string.
list            List all installed applications
listModules     List the modules in a specified application
options         Shows the options available, either for a given file, or in
                general.
publishWSDL     Publish WSDL files for a given application
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
updateAccessIDs Updates the user/group binding information with accessID
                from user registry for a given application
deleteUserAndGroupEntries
                Deletes all the user/group information for all
                the roles and all the username/password information for RunAs
                roles for a given application.
```

At this point we're interested in the `install` method

Output from the "help" method on the $AdminApp object

☐ To get help on the install method, issue the following command:

```
$AdminApp help install
```

You will 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.
```

A bit misleading ... arguments are *not* separated by a comma, but rather by a space

Note

Help output for the install method of $AdminApp object

So this is saying that the install method takes a pointer to the EAR file to be installed, and "options." What are those options? There are two answers to that:

1. Options applicable to *all* EAR files, regardless of what's inside the EAR
2. Options applicable to a given EAR file *based on what's inside that EAR file*

☐ To get a listing of the options applicable to all EAR files (the first answer in the two listed above), issue the following command:

```
$AdminApp options
```

You'll see a listing that looks like this:

```
WASX7105I: The following options are valid for any ear file:

server
cluster
cell
node
installdir
was.install.root
configroot
appname
verbose
contextroot
update
update.ignore.old
update.ignore.new
depl.extension.reg
defaultbinding.datasource.jndi
defaultbinding.datasource.username
defaultbinding.datasource.password
defaultbinding.cf.jndi
defaultbinding.cf.resauth
defaultbinding.ejbjndi.prefix
defaultbinding.virtual.host
defaultbinding.force
defaultbinding.strategy.file
```

Please note ... this does NOT mean all these options must be specified to install an application. For this set of exercises we'll specify only `server` and `node`

General options for the `install` method of `$AdminApp` object

Notes:

- The help output doesn't specify this, but each option has a dash on the front end. So the "server" option is specified: `-server` followed by its arguments.
- But that is *not* a complete listing of all the possible options. Rather than spit out all the options, the `$AdminApp` object has a facility to tell you what objects relate to a particular EAR file. It will interrogate the EAR file's deployment descriptors and then return the install method options that could apply. The format of that command is:

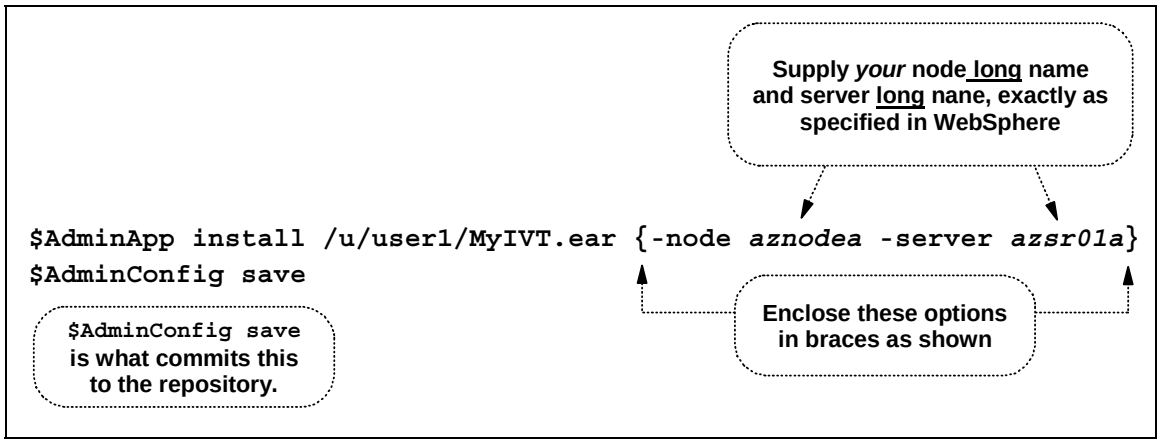
  ```
  $AdminApp options /<path>/<file_name>.ear
  ```

  Again, the listing that comes back does *not* mean all those options must be used to install the application. Some (or perhaps most) of that information is already specified in the EAR file's deployment descriptors. But if you want to override a particular value -- say, for example, a servlet's reference to an EJB, or the virtual host to which a web module is mapped -- then you may specify those options to accomplish the override.

☐ Issue the command **`exit`** to quit the WSADMIN prompt. That'll put you back at the OMVS prompt, which is where the next set of instructions assumes you'll be.

Install MyIVT using a simple script file

- ☐ Create a file in the `/u/user1` directory called `lesson4a.jacl`
- ☐ Edit the file and code the following:



`lesson4a.jacl` - *Two simple commands used to install and commit MyIVT to configuration*

??? What's the purpose of the braces? It groups up the `-node` and `-server` options as a single argument as input to the `install` method. That method -- `install` -- calls for *two* arguments only: EAR file name and "options." By enclosing the options in braces, it becomes a single argument for method `install`. See "Jacl lists -- an important way of providing options to WSADMIN commands" on page 30 for more on this.

- ☐ Invoke the script using the following command (*all on one line*):

```
./wsadmin.sh -javaoption -Dscript.encoding=Cp1047
     -conntype SOAP -host <host> -port <port> -f /u/user1/lesson4a.jacl
```

Notes:
- The host and port values would *not* be enclosed in a brackets. Those are there simply to show where you supply your values. An example of how it might look in real life is:

  `-host wsc3.washington.ibm.com -port 15510`
- Add `-user` <*userid*> and `-password` <*password*> if the server has global security enabled.

Two things will indicate success:

- `ADMA5013I: Application My_IVT_Application installed successfully.`
- A return to the command prompt

Important If you have a Network Deployment configuration, *do not try to start your application*. It is installed into the "master configuration" but it is not yet synchronized to the node. We will cover synchronizing to nodes under "Synchronizing changes with nodes" on page 49. For now, we're only focusing on the task of installing the application, *not* synchronizing and using it.

- ☐ Go to the Admin Console and see My_IVT_Application under "Applications"

Note: If you were already logged onto the Admin Console when you invoked WSADMIN, you'll have to log out and log back in to see the application.

- ☐ Log out of the Admin Console. (See "Important: Do NOT use WSADMIN and Admin Console at the same time" on page 12.)

Uninstall MyIVT

Now let's turn around and uninstall that application. This is a far easier process.

- ☐ Create a file in the `/u/user1` directory called `lesson4b.jacl`
- ☐ Edit the file and add two lines:

```
$AdminApp uninstall My_IVT_Application
$AdminConfig save
```

`lesson4b.jacl` - *Uninstalling the application*

Note: The `$AdminConfig save` does not appear to be strictly required. It appears to save okay without it. But adding it doesn't seem to hurt, and it keeps you in practice when installing applications, where it *does* appear to be required.

??? When uninstalling an application, you must use the "Application Name" as known by WebSphere. The EAR file name is meaningless.

- ☐ Invoke the script using the following command (*all on one line*):

```
./wsadmin.sh -javaoption -Dscript.encoding=Cp1047
     -conntype SOAP -host <host> -port <port> -f /u/user1/lesson4b.jacl
```

Two things will indicate success:

- `ADMA5106I: Application My_IVT_Application uninstalled successfully.`
- A return to the command prompt

Determining what task options are applicable to MyIVT.ear

In that last exercise the only options we used were `-node` and `-server`. We mentioned that one of the reasons we used the `MyIVT.ear` application was the relative simplicity of it. But "real" applications will require other mappings to things like data resources and virtual hosts. What options apply to the `MyIVT.ear` application, based on the deployment descriptors? Let's find out.

- ☐ Issue the following command:

```
./wsadmin.sh -conntype none
```

That will take you to the WSADMIN prompt.

- ☐ Issue the following command:

```
$AdminApp options /u/user1/MyIVT.ear
```

You'll get back a long list of options applicable to `MyIVT.ear`. This list will include all the *general* options (as illustrated in the figure labeled "General options for the `install` method of `$AdminApp` object" on page 39) as well as other options implied by the contents of the deployment descriptors.

??? How can we tell what the syntax is for any one of those given options? That's next.

Finding out more about a particular task option

- ☐ To get help on a given option -- say, `MapWebModToVH` -- issue the following command:

```
$AdminApp taskInfo /u/user1/MyIVT.ear MapWebModToVH
```

You'll get the following:

```
MapWebModToVH: Selecting Virtual Hosts for Web Modules

Specify the virtual host where you want to install the Web modules
contained in your application. Web modules can be installed on the
same virtual host or dispersed among several hosts.

WASX7348I: Each element of the MapWebModToVH task consists of the
following 3 fields: "webModule", "uri", "virtualHost".
Of these fields, the following may be assigned values: "virtualHost"
and the following are required: "virtualHost"

The current contents of the task after running default bindings are:
webModule: My_IVT_Webapp_Display_Name
uri: MyIVTWebApp.war,WEB-INF/web.xml
virtualHost: default_host
```

Provides feedback on what value is found in the EAR file for this option

$AdminApp taskInfo method will return information about a particular task option

Note: Unfortunately, that's not for all the options, only the following:

```
MapRolesToUsers, MapRunAsRolesToUsers, CorrectUseSystemIdentity,
BindJndiForEJBNonMessageBinding, BindJndiForEJBMessageBinding,
MapEJBRefToEJB, MapResRefToEJB, MapResEnvRefToRes,
DataSourceFor10EJBModules, DataSourceFor20EJBModules,
DataSourceFor10CMPBeans, DataSourceFor20CMPBeans, MapWebModToVH,
MapModulesToServers, EnsureMethodProtectionFor10EJB,
EnsureMethodProtectionFor20EJB, CorrectOracleIsolationLevel,
BackendIdSelection, AppDeploymentOptions, EJBDeployOptions,
GetServerName, WSDeployOptions
```

??? That doesn't really tell us what the syntax is, does it? Continue on ...

☐ Go the InfoCenter, do a search on "MapWebModToVH" and you'll get the following:

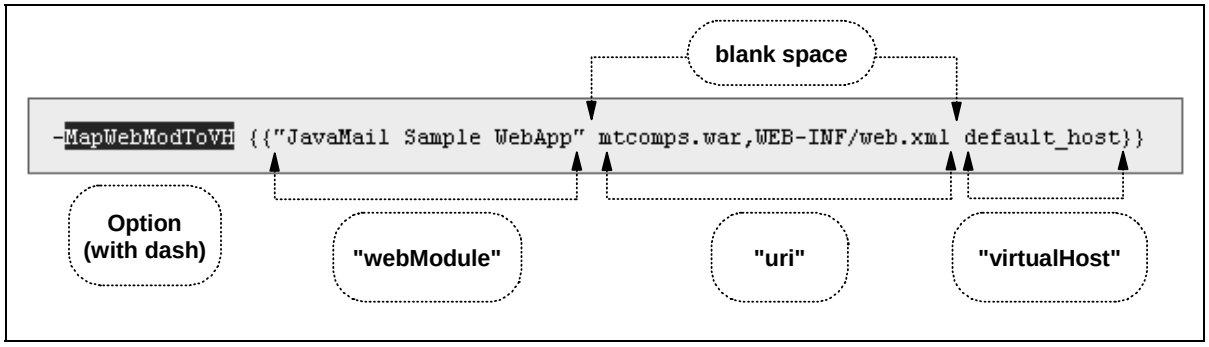
```
-MapWebModToVH {{"JavaMail Sample WebApp" mtcomps.war,WEB-INF/web.xml default_host}}
```

InfoCenter information on MapWebModToVH task option syntax (in Jacl)

Note: The InfoCenter is at URL:

`http://publib.boulder.ibm.com/infocenter/wasinfo/index.jsp`

Using the information about the `MyIVT.ear` file derived from the previous command, here's what each field in the InfoCenter example stands for:



What each field in the -MapWebModToVH option syntax relates to

So for the `MyIVT.ear` application the `-MapWebModToVH` option string would be:

```
{{"My_IVT_Webapp_Display_Name" MyIVTWebApp.war,WEB-INF/web.xml default_host}}
```

Notes:
- Your eyes are not deceiving you ... two braces to open the string, two braces to close the string. The outer braces enclose the entire argument list, the inner braces the three components shown. If `MapWebModToVH` had other arguments, they'd be enclosed in a separate set of "inner braces," inside the "outer braces."
- That process -- using `taskInfo` on the EAR to determine the applicable options, then using the InfoCenter to determine the syntax -- can be used for any of the options.

☐ Enter the command **`exit`** to return to the OMVS command prompt.

Installing MyIVT.ear and mapping to a different Virtual Host

In this exercise we will use the `-MapWebModToVH` option to map the web application inside `MyIVT.ear` to something other than the default virtual host.

☐ Go into the Admin Console and create a new virtual host. Call it something like `WSC_host`.

Note: Or any value you wish. It cannot contain blank spaces.

Admin Console Navigation
Environment
Virtual Hosts
New button

Location in the Admin Console navigation where new virtual host is created

☐ Before saving to the "Master Configuration," create a host alias for this virtual host.

- Click on the link that represents your new virtual host
- Click on "Host aliases"
- Click on "New" button
- Provide a "Host name" of a single asterisk (*)
- Provide a port value equal to the HTTP port of your application server
- Click on "OK"

☐ Save to the Master Configuration

Note: If you want to actually *use* this new virtual host -- as opposed to simply mapping an application to it -- then you will have to stop and restart your application server.

☐ Log out of the Admin Console. (See "Important: Do NOT use WSADMIN and Admin Console at the same time" on page 12.)

☐ Create a file `/u/user1/lesson4c.jacl`

☐ Code the following (*see notes below*):

```
$AdminApp install /u/user1/MyIVT.ear {-node <node> -server <server> -MapWebModToVH
      {{"My_IVT_Webapp_Display_Name" MyIVTWebApp.war,WEB-INF/web.xml WSC_host}}}

$AdminConfig save
```

`lesson4c.jacl` - *Uninstalling the application*

Notes:
- The entire `$AdminApp` line must be coded *on one line* in the Jacl script. In the example above, that's both the first and second lines ... all on one line in `lesson4c.jacl`
- There's one blank space between `-MapWebModToVH` and the opening `{{"My_IVT`... of the option string.
- Supply *your* node and server values for <*node*> and <*server*>
- If the virtual host you created earlier is anything *other* than `WSC_host`, then supply that value at the very end of the option string.
- We'll show you a way to manage the length of those lines in the next exercise, after we uninstall MyIVT.

☐ Invoke the script using the following command (*all on one line*):

```
./wsadmin.sh -javaoption -Dscript.encoding=Cp1047
      -conntype SOAP -host <host> -port <port> -f /u/user1/lesson4c.jacl
```

☐ If you coded `lesson4c.jacl` correctly, it should result in the application being installed and mapped to the virtual host.

☐ Log back into the Admin Console and verify that the application is indeed mapped to the new virtual host:

Admin Console Navigation
- **Applications**
  - **Enterprise Applications**
    - **<My_IVT_Application> link**
      - **Map virtual host for web modules**

Where you can validate (and change) the virtual host to which a web module is mapped

You should see that it's mapped to `WSC_host` (or your value, if you used a different virtual host name)

Important If you have a Network Deployment configuration, *do not try to start your application*. It is not yet synchronized to the node. We will cover synchronizing to nodes under "Synchronizing changes with nodes" on page 49.

Uninstall application in preparation for next exercise

☐ Re-use `lesson4b.jacl`, which uninstalled the application earlier.

```
./wsadmin.sh -javaoption -Dscript.encoding=Cp1047
   -conntype SOAP -host <host> -port <port> -f /u/user1/lesson4b.jacl
```

Using Jacl variables to construct the long command line

☐ Create a file `/u/user1/lesson4d.jacl`

☐ Code the following:

```
set ear     "/u/user1/MyIVT.ear"
set node    "<node>"
set server  "<server>"
set WebMod  "\"My_IVT_Webapp_Display_Name\""
set uri     "MyIVTWebApp.war,WEB-INF/web.xml"
set VH      "WSC_host"
# ------------------------------------------------------------------
set VHlist  "$WebMod $uri $VH"
set VHopts  [list $VHlist]
set options [list -node $node -server $server -MapWebModToVH $VHopts]
# ------------------------------------------------------------------
$AdminApp install $ear $options
$AdminConfig save
```

1 2 3 4 5

```
[ = x'AD' EBCDIC
] = x'BD' EBCDIC
```

lesson4d.jacl - *Construction of long command line using Jacl*

1 Code *your* node and server long names in place of <node> and <server>

2 The sequence \" is a way to get double-quote marks into the actual variable. Recall that the web module portion of the MapWebModToVH option string was enclosed in double quotes.

3 Why are we putting the three variables into a single variable? It turns out Jacl does an odd thing if a variable included on a list string contains double-quotes: it encloses that variable in its own set of braces. If we were to code [list $WebMod $uri $VH] that would result in

{**("webmod"**} uri vh}

when what we really want is:

{"webmod" uri vh}

By placing the three components first into a regular string variable and then using list, we can then use the list function to enclose the three in a single set of braces.

4 Here we pace the variable $VHlist into a set of single braces using the Jacl list function. A new variable is created: VHopts.

5 Here we create the variable options which will be a list (enclosed in braces). We're placing into this list the variable $VHopts, *itself a list*. An interesting thing happens: Jacl wraps the contents of $VHopts -- which at this point is {"webmod" uri vh} -- in another set of braces, creating the necessary {{"webmode" uri vh}} string of characters.

☐ Issue the following command:

```
./wsadmin.sh -javaoption -Dscript.encoding=Cp1047
      -conntype SOAP -host <host> -port <port> -f /u/user1/lesson4d.jacl
```

Important If you have a Network Deployment configuration, *do not try to start your application*. It is not yet synchronized to the node. We will cover synchronizing to nodes under "Synchronizing changes with nodes" on page 49.

Jacl script that installs or uninstalls based on passed in parameter

☐ Create a file /u/user1/lesson4e.jacl

☐ Code the following:

```
set parm [lindex $argv 0]
if { !($parm == 1 || $parm == 2) } then {
 puts stdout "Must supply either 1 for install or 2 for uninstall"
 exit
}
# EAR FILE, OPTIONS ------------------------------------------------
set ear      "/u/user1/MyIVT.ear"
set name     "My_IVT_Application"
set node     "<node>"
set server   "<server>"
set opts     [list -node $node -server $server]
# INSTALL/UNINSTALL ------------------------------------------------
if { ($parm == 1) } then {
 puts stdout "Installing $name"
 $AdminApp install $ear $opts
 $AdminConfig save
} else {
 puts stdout "Uninstalling $name"
 $AdminApp uninstall $name
 $AdminConfig save
}
# -----------------------------------------------------------------
puts stdout "All done."
```

1
2
3
4

[= x'AD' EBCDIC
] = x'BD' EBCDIC

lesson4e.jacl - *Construction of long command line using Jacl*

1 Variable `parm` set to the first parameter passed in. (If more than one passed in, others ignored.) Check made to see if value is either `1` or `2`. Anything else results in warning message and exit from script.

2 Set `<node>` and `<server>` equal to the long name values for your environment.

3 If the parameter was `1` then the application is installed.

4 Otherwise the application is uninstalled.

Note: You may wonder what would happen if a parameter of 1 was passed in when the application was already installed in the environment. WSADMIN would simply kick the installation back with warning message about application already being in the environment. This Jacl script does no checking to see if that's the case, though it *could*.

☐ For the first test, see what happens when no parameters are passed in. Issue the following command:

```
./wsadmin.sh -javaoption -Dscript.encoding=Cp1047
     -conntype SOAP -host <host> -port <port> -f /u/user1/lesson4e.jacl
```

You should get back the warning message with information about what parameters are valid.

??? The same thing would happen if you passed in 3, or 5, or "Fred," or any parameter other than 1 or 2.

☐ The application should still be in your environment from the previous lesson. So use parameter `2` to *uninstall* the application. Issue the following command:

```
./wsadmin.sh -javaoption -Dscript.encoding=Cp1047
     -conntype SOAP -host <host> -port <port> -f /u/user1/lesson4e.jacl 2
```

You should see the "All done" message.

Concluding points on this lesson

We explored a little of the `$AdminApp` object. As you saw when you issued the `$AdminApp help` command, there was quite a few methods on that object. And under just one of those methods -- `install` -- there was 20 or more options. There are three levels of complexity to all of this that simply takes time to get used to:

1. *The methods, tasks and options* -- we saw how to drive down and get the syntax for the `-MapWebModToVH` task under the `install` method. You can use that to get the syntax for any of the methods.
2. *When to use what task for what purpose* -- let's say you have an EAR file and you know that you want to remap the JNDI name of a session bean. Which task on the `install` method do you use? (The answer is `BindJndiForEJBNonMessageBinding`, but determining that was a matter of taking a few educated guesses and exploring.)
3. *Sorting out the exact syntax of the task* -- and for this there are *two* levels of complexity:
   - *What values to code for the various task arguments* -- for example, the value for the web module's "uri" value in the `-MapWebModToVH` option was:

     ```
     MyIVTWebApp.war,WEB-INF/web.xml
     ```

     But you'd only know that if you already knew it, or you used the `taskInfo` mechanism to have WSADMIN tell you what the present value already is (see "Finding out more about a particular task option" on page 21).
   - *Opening and closing braces, dashes, etc*. -- getting comfortable enough with the syntax to know when a dash is required and when braces are necessary. The help function of WSADMIN is a bit weak in this area. The InfoCenter provides much better examples.

     Note: And this is made more complex when you fold the WSADMIN commands into more complex Jacl scripting. There it becomes necessary to nest `list` functions. It can get very confusing very quickly.

The good news is this: simple EAR files are relatively simple to install. And by doing that you can gain enough practice to then branch into more complex things.

Lesson 5: The $AdminConfig Object

Note: WSADMIN is most useful for repetitive tasks where consistency of operation is the objective. Things done just once and not again and again -- as is the case with configuration changes -- might more easily be done through the Admin Console. Nevertheless, we'll cover `$AdminConfig` here so you can see how it works.

The `$AdminConfig` object is used to create, remove or modify "configuration objects," such as servers, virtual hosts, clusters, etc.

Note: While an application is part of the configuration, they're handled by the `$AdminApp` object, as we just looked at in the "Lesson 4: Installing an Application using $AdminApp Object" section, starting on page 35.

A little background on `$AdminConfig`

Before jumping into the exercises, let's look at a few things first:

What methods are on this object?

Quite a few. You can get a listing of the methods by invoking the `help` method of the `$AdminConfig` object:

```
./wsadmin.sh -conntype none -c '$AdminConfig help'
```

You get a listing that looks like this:

```
attributes                          help
checkin                             list
convertToCluster                    listTemplates
create                              modify
createClusterMember                 parents
createDocument                      queryChanges
installResourceAdapter              remove
createUsingTemplate                 required
defaults                            reset
deleteDocument                      save
existsDocument                      setCrossDocumentValidationEnabled
extract                             setSaveMode
getCrossDocumentValidationEnabled   setValidationLevel
getid                               show
getObjectName                       showall
getSaveMode                         showAttribute
getValidationLevel                  types
getValidationSeverityResult         validate
hasChanges
```

Methods on the `$AdminConfig` object, as listed by the `help` method

How many different "configuration types" exist?

"Types" are things like Web Containers, HTTP Transport, Java Virtual Machine, Resource Adapter -- in other words, pieces of the overall configuration puzzle. There are many "types." You can get a listing of all the "types" supported by the `$AdminConfig` object by issuing the command:

```
./wsadmin.sh -conntype none -c '$AdminConfig types'
```

You'll get a list of "types" 225 items long.

??? What's the message? The message is there's a whole bunch of things the `$AdminConfig` object can operate upon. The objective of this document is to merely familiarize you with the process so you can then explore on your own. In other words, we won't go over all 225 types here.

Does `$AdminConfig` require a connection to a server process?

Not necessarily. The `$AdminConfig` option can work against the repository without being connected to a server process. But the general rule of thumb mentioned back in the `$AdminApp` section is still appropriate here:

If the Administrative Appl is *running*	use `-conntype SOAP -host <host> -port <port>`
If the Administrative Appl is *stopped*	use `-conntype none`

Note: An exception to this is if you're using `$AdminConfig` to simply list out what's in the configuration repository. Then `-conntype none` is fine. But if you're looking to *change* the repository, then use the rule of thumb above.

Caution: z/OS is different type of environment from distributed

The `$AdminConfig` object can be used to do things like create a new server. In the z/OS world, however, *more* work may be required to permit that server to actually operate. For example, any required RACF work is outside the scope of WSADMIN. The examples in the InfoCenter show the creation of servers and the starting of those servers all within the same Jacl script. That works on the distributed systems, but not z/OS. Be aware of this distinction when WSADMIN is operating against WebSphere Application Server running on a z/OS system.

Note: It *is* possible to create a batch job that invoked WSADMIN to create a server, then invoked a series of RACF commands for the new server, then invoked WSADMIN again to start the server. So it's not impossible to automate the process. But it will involve things outside WSADMIN.

Synchronizing changes with nodes

We saw in the previous lesson that in a Network Deployment configuration, changes made with WSADMIN through the Deployment Manager server process are saved only to the "master configuration." Just because you used `$AdminConfig save` does not mean the changes are synchronized to the nodes.

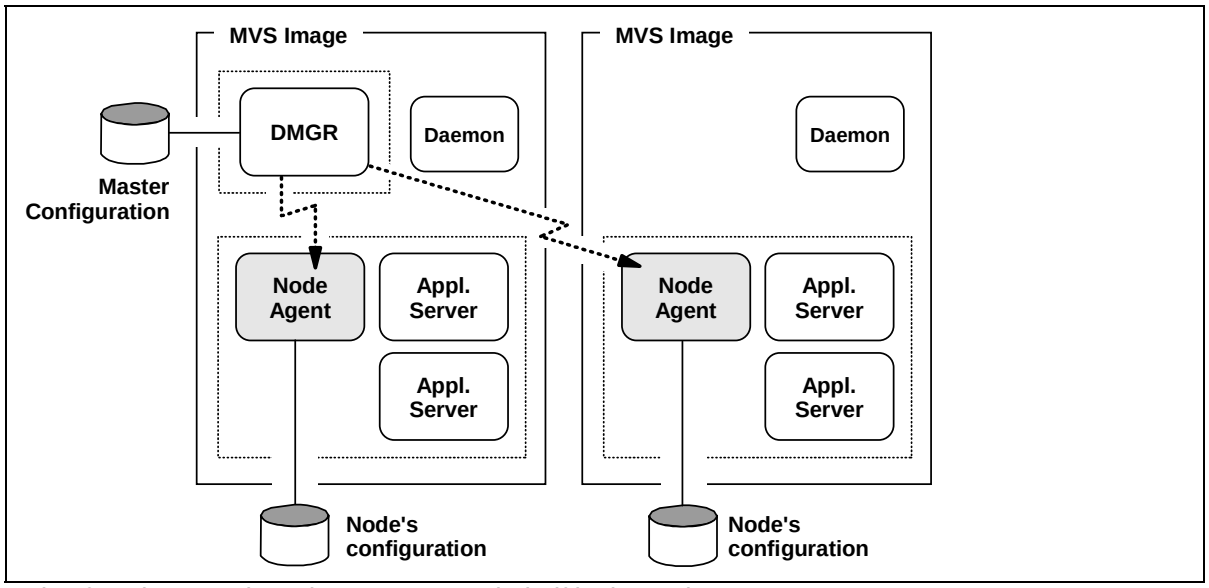
When are changes synchronized? It depends:

Initiated by WSADMIN script	WSADMIN has a facility that prompts a Node Agent to come and get the changes that have just been made to the "master configuration." That process is under "Initiating synchronization using WSADMIN" on page 50.
Initiated by Node Agent	Each node agent has configuration settings that determine how frequently it will contact the Deployment Manager and request changes. Those configuration settings are documented under "Node Agent configuration settings that affect synchronization intervals" on page 51.

Important: Synchronization *does not apply* to a Base Application Server node. By definition that has no Deployment Manager, no Node Agents, and only one node.

Node synchronization overview

In a Network Deployment configuration the Deployment Manager is where the administrative application runs. The administrative application maintains the "master configuration" in the HFS of the Deployment Manager. Each application server node has a Node Agent, which acts on the Deployment Manager's behalf in that node. Each node maintains a copy of the configuration in its HFS:



Role of Node Agent in acting on DMGR's behalf in the node

Note: This picture shows three HFS symbols, but that's not to say three different HFS files systems is required. One single HFS file system can be used (if shared HFS is available), or separate HFS file systems can be employed. The point is really that the DMGR's has a separate *directory structure* from the nodes, and each node has its own directory structure. Where the directory structure resides is irrelevant to this discussion.

Synchronization occurs on a node-by-node basis. That means that if you have two nodes (as pictured above), then "synchronization" implies two actions: synchronization to the first node, then synchronization to the other. Why that's important becomes clear in the next section where we show the WSADMIN command used to initiate synchronization. Here's a clue: the command is specific to a node. If you have four nodes, and you wish to synchronize the entire cell, it implies issuing *four* WSADMIN commands.

Initiating synchronization using WSADMIN

It's important to introduce at this point in the document the mechanism used to initiate synchronization through WSADMIN. But the mechanism is fairly complex. So for now, you will simply have to trust that is how it is done:

Value returned by `$AdminControl completeObjectName` command

```
WebSphere:platform=common,cell=g5cell,version=5.0,name=nodeSync,
  mbeanIdentifier=nodeSync,type=NodeSync,node=g5nodec,process=nodeagent
```

```
set var [$AdminControl completeObjectName type=NodeSync,node=g5nodec,*]
$AdminControl invoke $var sync
```

Long name of the node

Commands used to invoke synchronization for a node

What this is doing is retrieving the unique object name of the `NodeSync` Mbean of the Node Agent, then using that unique name to invoke the `sync` method of it.

Key Points:
- Initiating synchronization involves issuing *two* commands: one to get the unique name, one to invoke the synchronization
- This must be done for *each node* you wish to synchronize

Programmatically synchronizing with every node in the cell

What we just showed is a way to initiate synchronization with just one node. Let's say you want to synchronize your changes with *all the nodes*. How is that accomplished? By querying for a list of the nodes in the cell, then looping through the list and synchronizing each node:

```
set node_ids [$AdminConfig list Node] 1
foreach node $node_ids {
 set node_name [$AdminConfig showAttribute $node name] 3
 set nodeSync  [$AdminControl completeObjectName type=NodeSync,node=$node_name,*] 4
2  5 if { !($nodeSync=="") } then {
  $AdminControl invoke $nodeSync sync 6
 }
}
```

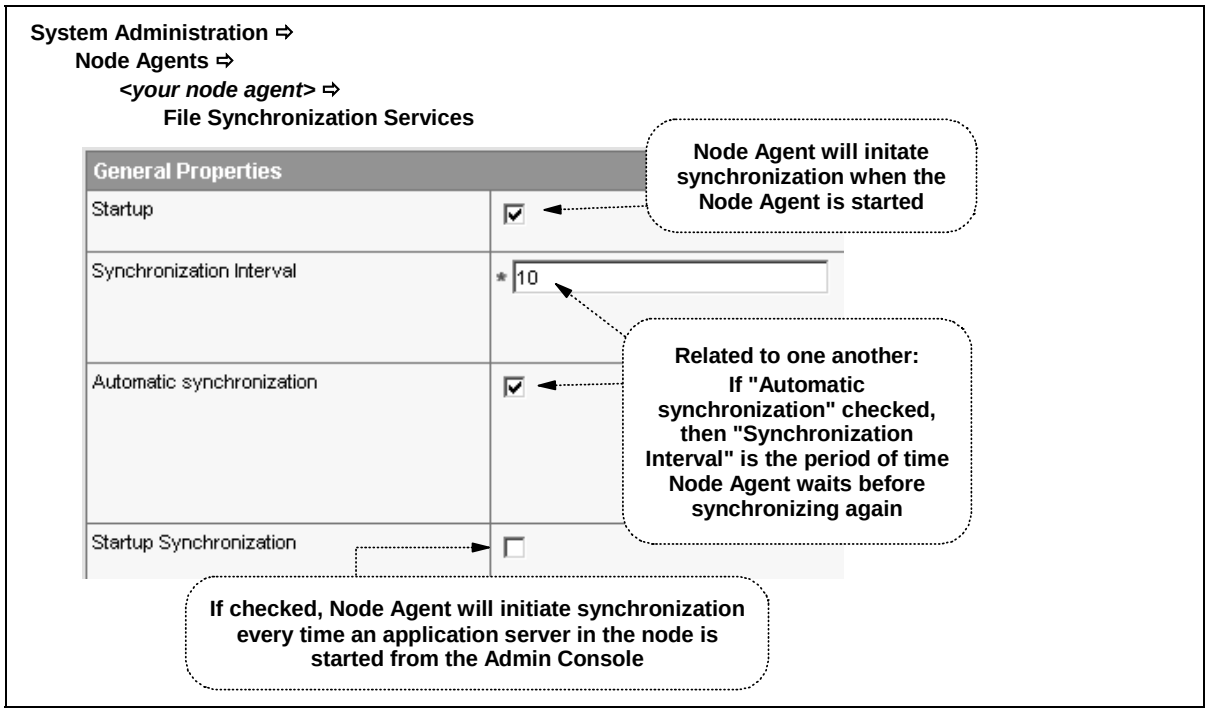
Synchronizing all the nodes in a configuration cell

??? That script is provided in the sample file `synch_all.jacl`, in the "Lesson 5" section.

1. Using the `list` function, all the nodes in the configuration are placed into a list variable called `node_ids`. This would include all the application server nodes and the Deployment Manager node. The value placed into the variable is the "config ID" of the node, which includes *more* than just the node name.
2. The `foreach` function loops through all the elements of the list variable `$node_ids`. On each iteration it places the element into another variable; in this example the variable `node`.
3. Here the node long name is extracted out of the config ID.
4. The "complete object name" of the node synchronization Mbean is extracted for that node.
5. A check is made to see if the value of `$nodeSync` is a null string. This is done to catch the Deployment Manager's node, which will return a null value for `$nodeSync`. If this check wasn't in place, the script would throw an error when the next line is invoked when the variable `$nodeSync` was null.
6. Finally, we synch to that node.

Node Agent configuration settings that affect synchronization intervals

Through the Admin Console you may change the settings that affect how frequently the Node Agents initiate synchronization:



A Node Agent's "File Synchronization Services" settings

So by default (the settings shown in the picture above), the following would occur:

- Every time the Node Agent starts up it will contact the Deployment Manager and request the updates to the master configuration.
- Every 10 minutes the Node Agent will "wake up" and request from the Deployment Manager the updates to the master configuration
- No synchronization will take place when individual servers are started up.

???	Would it be possible to avoid kicking off synchronization in WSADMIN and use this aspect of WebSphere? Yes, you could set these values lower and simply wait until synchronization occurs. But invoking synchronization at the time of update insures real-time access to the changes.

Exploring the VirtualHost type

We'll start with `VirtualHost` because it's something most of us know something about, and because it's something we can create and delete without impacting other things in our WebSphere environment.

How can we know that VirtualHost is a type?

By driving the `types` method of the `$AdminConfig` object. Do the following:

☐ Open an OMVS prompt. "su" to the WebSphere Administrator ID and change directories to the `/<config_root>/DeploymentManager/bin` directory.

☐ Invoke the following command:

```
./wsadmin.sh -conntype none -c '$AdminConfig types'
```

What you'll get back is a *long* listing, which includes:

```
  :
VariableMap
VariableSubstitutionEntry
VirtualHost
WAS40ConnectionPool
WAS40DataSource
  :
```

Near the bottom of the long list is `VirtualHost` ... one of the "types"

One of the configuration "types" is `VirtualHost`

What's the structure of the VirtualHost type?

Another method on the `$AdminConfig` object called `attributes` will give us a hint. Do the following:

☐ From the `OMVS` prompt (or telnet session), invoke the following command:

```
./wsadmin.sh -conntype none -c '$AdminConfig attributes VirtualHost'
```

What you'll get back is this:

```
$AdminConfig attributes VirtualHost
"aliases HostAlias*"
"mimeTypes MimeEntry*"
"name String"
```

The type has three attributes

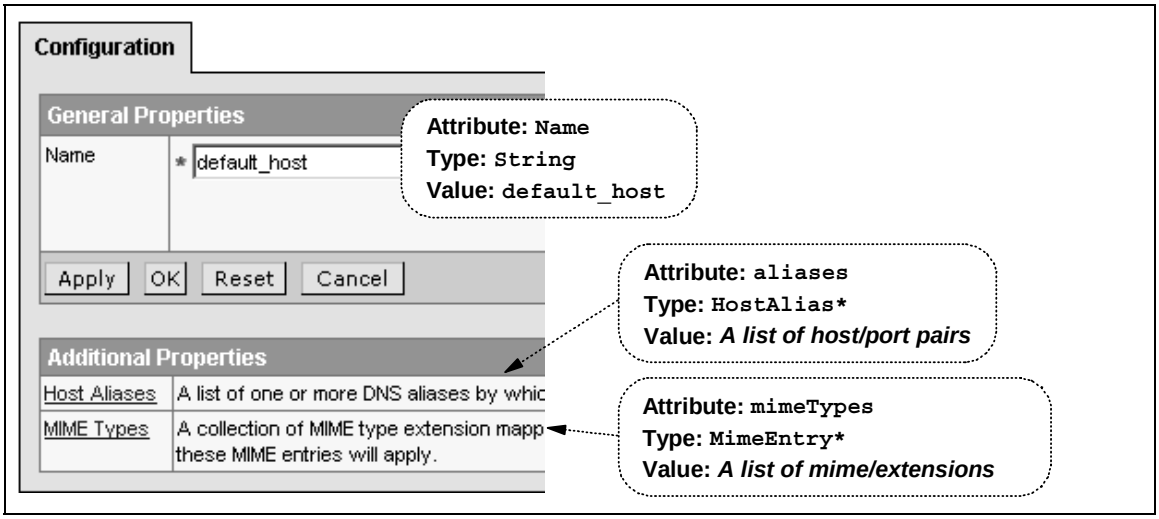
These are also types, and we need to dig deeper to find out the structure of them

"String" is just that ... a string of characters

These are keywords ... when we create a virtual host we'll use these exact words in the Jacl script

The attributes of the VirtualHost type

If you map this to what you see in the Admin Console, this makes some sense:



Attributes on VirtualHost type map to Virtual Host options in the Admin Console

We'll ignore the `MimeEntry` attribute and focus on the `HostAlias` attribute.

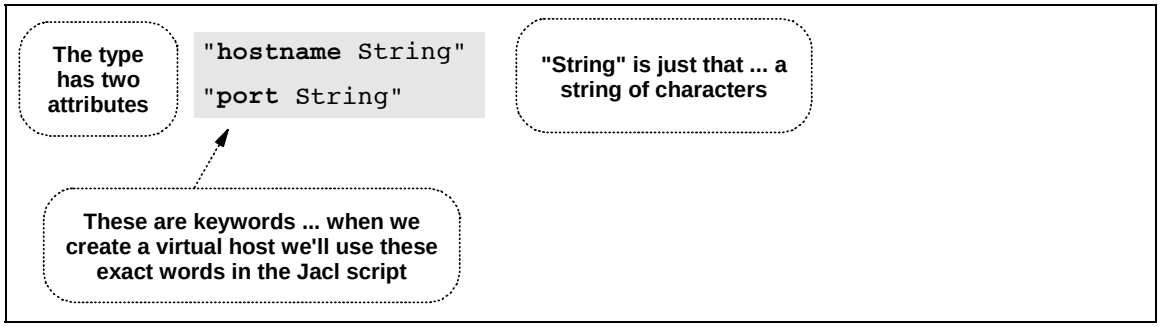
??? `MimeEntry` is nearly identical in *concept* to `HostAlias`. Different keywords used, but same notion of two name/value pairs used to construct an entry. Few people think about mime types, but everyone has to worry about `HostAlias` to make things work.

What's the structure of the `HostAlias` type?

- We just discovered that `HostAlias` is itself a configuration type. (In fact, if you look at the output from the `$AdminConfig` types command, you'll see `HostAlias` listed there.) That means we can run the `attributes` method against the type `HostAlias`:

```
./wsadmin.sh -conntype none -c '$AdminConfig attributes HostAlias'
```

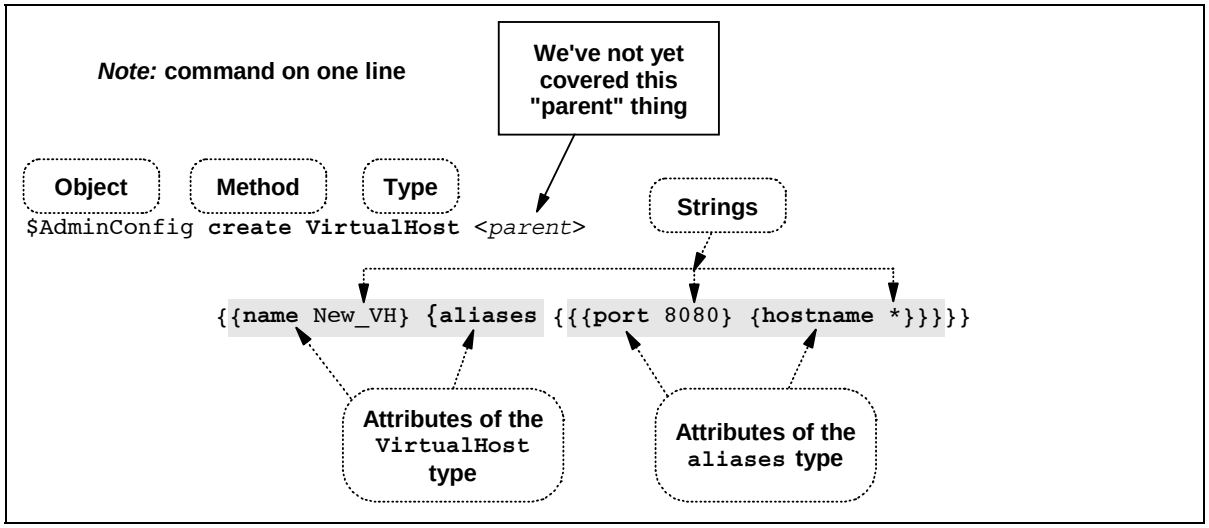
And what we get back is the following:



Attributes on the HostAlias configuration type

What's the value in what we just did?

All that was in preparation for understanding how to construct the command to create a virtual host using WSADMIN. What we will eventually see is that the command string looks something like this:



Peek at what command looks like to create a new virtual host with one alias

By drilling down with the attributes method, we discovered what things are expected when creating the `VirtualHost` configuration type.

Note: Understanding what's expected on a command is half the battle. With that in your mind you'll be better prepared to make sense of the samples provided in the InfoCenter. Without this knowledge of "types" and "attributes" it would be difficult to discern what the samples were trying to do.

What's the minimum required to construct a VirtualHost configuration type?

As you've noticed, these configuration types may have many different attributes. Some of those attributes are themselves types which have their own attributes. Clearly there'll be a desire at times to understand what is the least complex way to create a configuration type.

Thankfully, the `$AdminConfig` object has a method called `required`. That method will tell you the required attributes of a given type.

☐ Issue the following command:

```
./wsadmin.sh -conntype none -c '$AdminConfig required VirtualHost'
```

You'll get back the following:

```
Attribute        Type
name             String
```

What this is telling you is that the minimum you can get away with when creating a `VirtualHost` type is the `name` attribute. The attributes `HostAlias` and `MimeEntry` are optional *at the time of creation*.

> **Note:** A virtual host with no host alias can be created, but an application mapped to that virtual host won't work. This highlights an important point: the `required` method is simply telling you the minimum to *create* the configuration type, not necessarily to use it.

Using WSADMIN to create a simple, no-alias Virtual Host

In this exercise we'll create a virtual host using the `create` method of `$AdminConfig`. We'll create it using only the `name` attribute. This is to keep things simple for now, and to introduce the notion of a type's "parent."

The `help` method of `$AdminConfig` will provide the following syntax for the `create` method:

Command Issued: `$AdminConfig help create`

Result Received: `Arguments: type, parent, attributes`

Determining the ID of the cell to provide as the "parent" for the create method

We know that in WebSphere Application Server for z/OS the cell name takes two forms: a *long* name and a *short* name. For the purpose of supplying the cell as a "parent" on the create method, we need something else: we need the "ID" of the cell.

We can use the `getid` method of `$AdminConfig` to fetch the ID of the cell from the configuration repository. Do the following:

☐ Create a file called `/u/user1/lesson5a.jacl`

☐ Code the following in the file:

```
[ = x'AD' EBCDIC
] = x'BD' EBCDIC
```

Provide *your* cell long name here

```
set cellid [$AdminConfig getid /Cell:g5cell/]
puts stdout $cellid
```

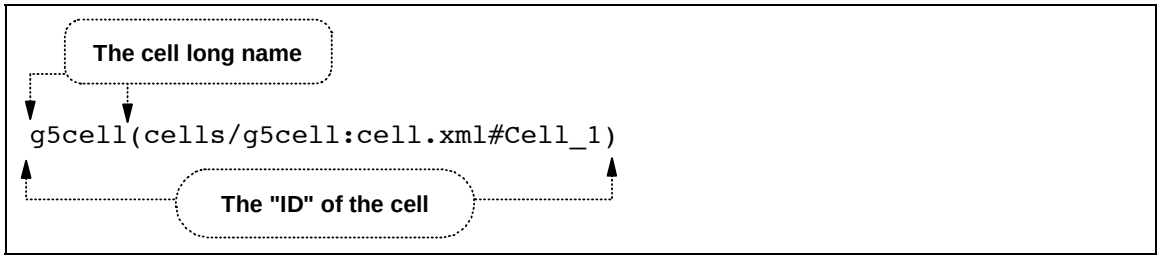
`lesson5a.jacl` - *Using getid method to determine ID of the cell*

> **Note:** The string `/Cell:g5cell/` is know as the "containment path" for the object. The containment path is delimited by slashes as shown.

☐ Issue the following command (all on one line):

```
./wsadmin.sh -javaoption -Dscript.encoding=Cp1047
     -conntype SOAP -host <host> -port <port> -f /u/user1/lesson5a.jacl
```

What you'll get back will look like this:



What comes back from the `getid` method when the cell ID is what we were after

Now we know the ID of the cell, which will serve as the "parent" when we create the Virtual Host:

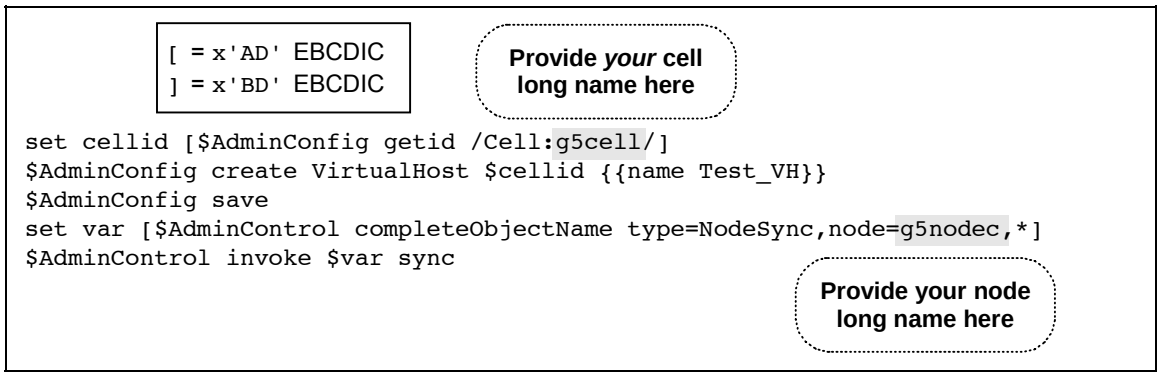
```
g5cell(cells/g5cell:cell.xml#Cell_1)
```

We could code that on the command itself, or pass it in as a variable. We'll pass it in as a variable.

Jacl script to get the cell ID, then create no-alias virtual host

☐ Create a file called `/u/user1/lesson5b.jacl`

☐ Code the following in the file:



`lesson5b.jacl` - *Creating a virtual host and providing cell ID as parent*

Notes:
- If you have a Base Application Server node, then delete the last two lines. Those invoke synchronization, which does not apply to a BaseApp node.
- If you have multiple nodes in a Network Deployment configuration, see "Programmatically synchronizing with every node in the cell" on page 51.

☐ Issue the following command (all on one line):

```
./wsadmin.sh -javaoption -Dscript.encoding=Cp1047
     -conntype SOAP -host <host> -port <port> -f /u/user1/lesson5b.jacl
```

If the script was without errors, it'll finish by returning you to the command prompt.

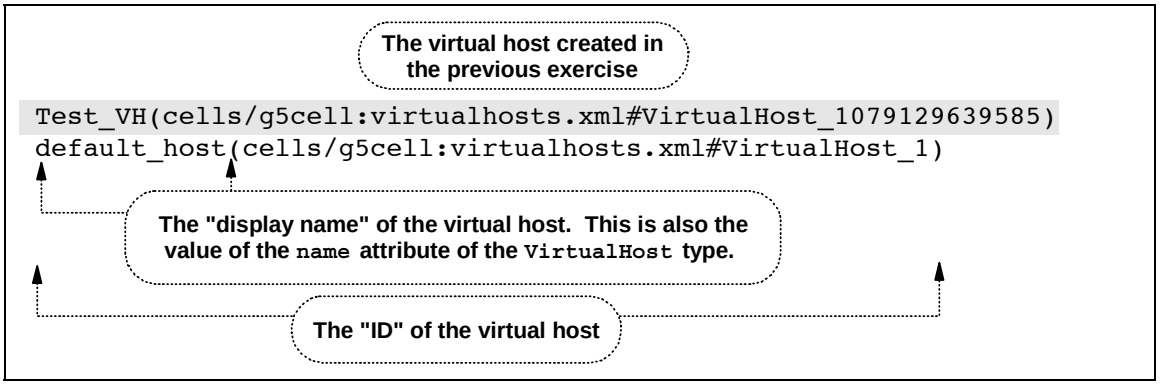
Listing the existing VirtualHost types (including your new one)

The `list` method on the `$AdminConfig` object will list out all the instances of a given type. Do the following:

☐ Issue the following command:

```
./wsadmin.sh -conntype none -c '$AdminConfig list VirtualHost'
```

This will list all the configuration objects in the repository of type `VirtualHost`. You will see in return a list from 1 to *n* items long, depending on how many different virtual hosts you have configured in your repository. The listing will look something like this:



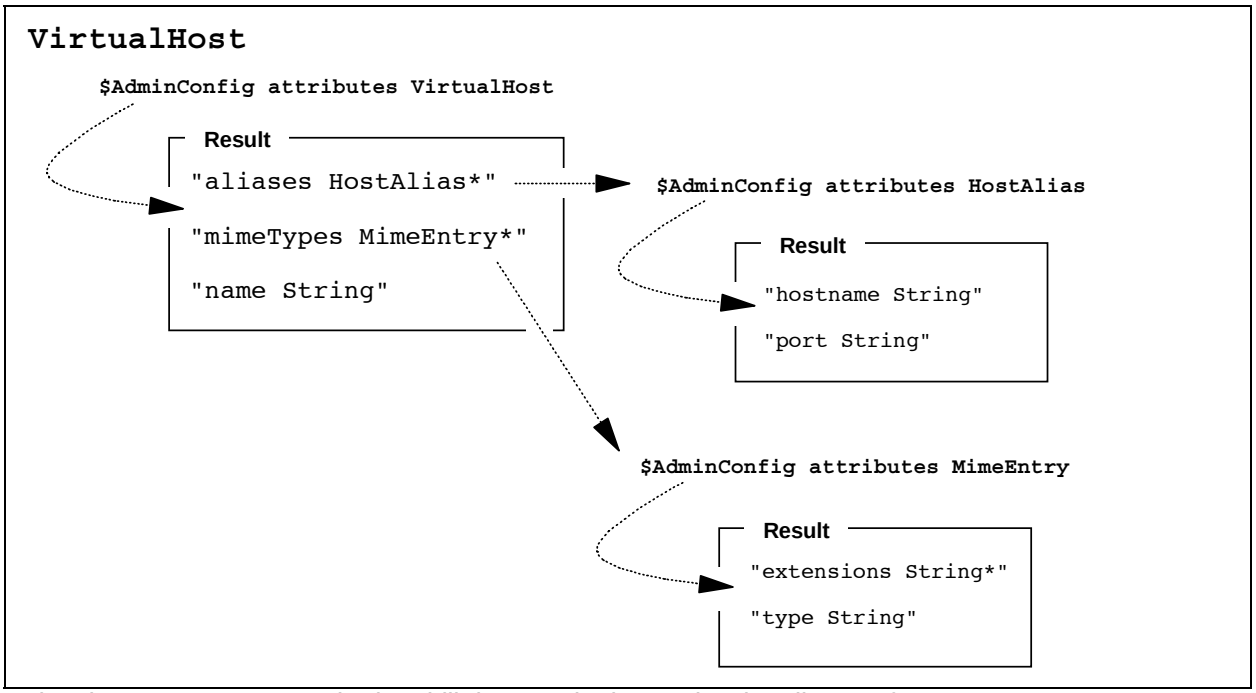
Output from `list` method when the type being listed is `VirtualHost`

??? The notion of the "ID" becomes very important later when we modify existing configuration objects. `$AdminConfig` will require the ID to uniquely identify the exact thing you're looking to modify. Typically what you'll see is a sequence of things in a Jacl script: 1) use WSADMIN functions to extract the ID value, 2) put the results in a variable, then 3) use the variable when constructing the command to modify the object.

Using WSADMIN to determine the values assigned to an existing virtual host

Now let's look at how methods on the `$AdminConfig` object can tell you about existing virtual hosts. This will set the stage for the creation of configuration elements programmatically; that is, querying the environment to get some information, storing that information into a variable, then using that variable to create something else in the repository.

Here's what we know about the `VirtualHost` configuration type:

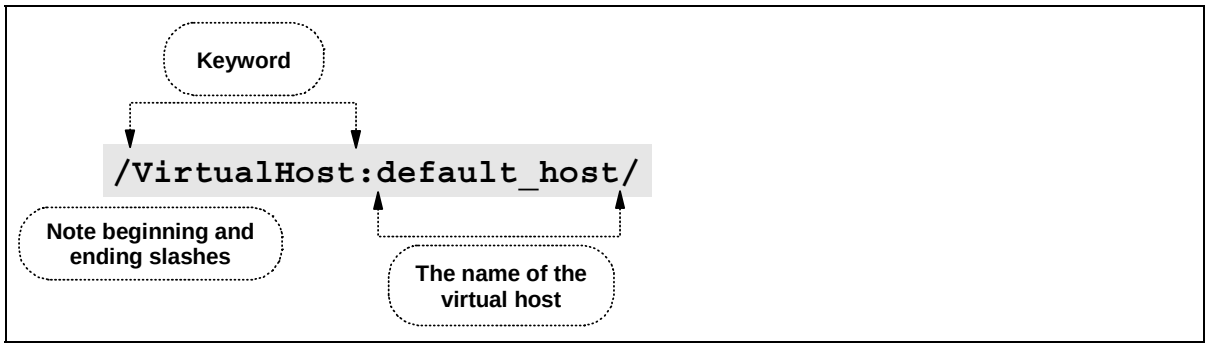


Using the `attributes` method to drill down to the lowest-level attributes of `VirtualHost`

From our use of the `$AdminConfig list VirtualHost` command, we know that we have a virtual host with a name of `default_host`. Can we use `$AdminConfig` to tell us what the attribute *values* are for `default_host`? Yes.

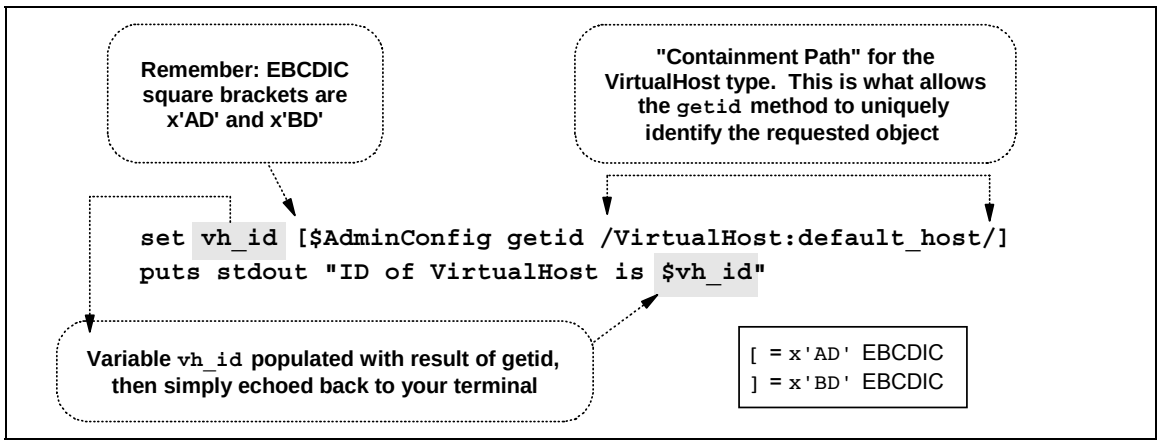
Using the getid method to place the config ID of default_host into a Jacl variable

The `getid` method of `$AdminConfig` will return the unique "config ID" of a configuration object. In this case we're interested in the config ID of the `default_host` virtual host. The `getid` method takes as an argument the "containment path" of the configuration object in question. The containment path for a virtual host looks like this:



Format of the "containment path" for the `VirtualHost` configuration object

- ☐ Create a file called `/u/user1/lesson5c.jacl`
- ☐ Code the following in the file



`lesson5c.jacl` - *Extracting the ID of the virtual host `default_host`*

- ☐ Issue the following command (all on one line):

```
./wsadmin.sh -javaoption -Dscript.encoding=Cp1047
     -conntype SOAP -host <host> -port <port> -f /u/user1/lesson5c.jacl
```

You should get a response that looks *something* like this:

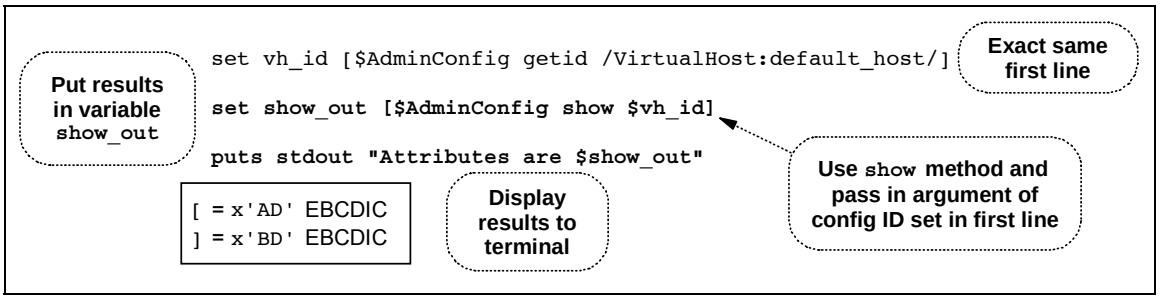
```
ID of VirtualHost is
        default_host(cells/g5cell:virtualhosts.xml#VirtualHost_1)
```

We now have the ID of the virtual host `default_host` in a Jacl variable. Now we can work with it.

Using the show method to display all the attributes held by the configuration object

The `show` method has a single argument: the "config ID" of the object. In the last exercise we captured the config ID for the virtual host `default_host` in the variable `vh_id`. Let's now use that with the `show` method.

- ☐ *Copy* the file `/u/user1/lesson5c.jacl` to file `/u/user1/lesson5d.jacl`
- ☐ Edit the file `lesson5d.jacl` and provide the following:



`lesson5d.jacl` - *Using `show` method of `$AdminConfig` to show all attributes of VH*

- ☐ Issue the following command (all on one line):

```
./wsadmin.sh -javaoption -Dscript.encoding=Cp1047
     -conntype SOAP -host <host> -port <port> -f /u/user1/lesson5d.jacl
```

Brace yourself ... you should see an output that looks like this:

```
Attributes are {aliases {(cells/g5cell:virtualhosts.xml#HostAlias_1)
(cells/g5cell:virtualhosts.xml#HostAlias_2) (cells/g5cell:virtualhosts.xml#HostAlias_3)
(cells/g5cell:virtualhosts.xml#HostAlias_4)
(cells/g5cell:virtualhosts.xml#HostAlias_1078935918161)}}
{mimeTypes {(cells/g5cell:virtualhosts.xml#MimeEntry_1)
(cells/g5cell:virtualhosts.xml#MimeEntry_2)
(cells/g5cell:virtualhosts.xml#MimeEntry_3) (cells/g5cell:virtualhosts.xml#MimeEntry_4)
(cells/g5cell:virtualhosts.xml#MimeEntry_5) (cells/g5cell:virtualhosts.xml#MimeEntry_6)
  :
  :
(cells/g5cell:virtualhosts.xml#MimeEntry_89) (cells/g5cell:virtualhosts.xml#MimeEntry_90)}}
{name default_host}
```

Many lines removed to save space in document

Output of `show` on `default_host`

??? The `default_host` virtual host has a very long list of `MimeEntry` types. 90 of them in total. That's a lot of output. So in the next exercise we'll show you how to trim that back by showing *just* the `HostAlias` type.

Using the showAttribute method to display a certain kind of attribute

Let's say you didn't want to see all that `MimeEntry` stuff. Let's say all you were interested in was the "hostname/port" aliases contained in the virtual host. Would it be possible to list just those? The answer is yes. The `showAttribute` method does exactly that.

- ☐ *Copy* the file `/u/user1/lesson5d.jacl` to file `/u/user1/lesson5e.jacl`
- ☐ Edit the file `lesson5e.jacl` and add provide the following:

```
set vh_id [$AdminConfig getid /VirtualHost:default_host/]

set show_out [$AdminConfig showAttribute $vh_id aliases]

puts stdout "Attributes are $show_out"
```

[= x'AD' EBCDIC
] = x'BD' EBCDIC

Changes to file are highlighted

lesson5e.jacl - *Using* `showAttribute` *method to display just* `aliases`

☐ Issue the following command (all on one line):

```
./wsadmin.sh -javaoption -Dscript.encoding=Cp1047
     -conntype SOAP -host <host> -port <port> -f /u/user1/lesson5e.jacl
```

What you'll get back is something far more manageable:

```
Attributes are
{(cells/g5cell:virtualhosts.xml#HostAlias_1)
 (cells/g5cell:virtualhosts.xml#HostAlias_2)
 (cells/g5cell:virtualhosts.xml#HostAlias_3)
 (cells/g5cell:virtualhosts.xml#HostAlias_4)
 (cells/g5cell:virtualhosts.xml#HostAlias_1078935918161)}
```

Five aliases held in default_host

This is the "config ID" for this particular host alias

Output to `showAttribute` *on just* `aliases` *attribute for* `default_host` *virtual host*

Note: Your output may be more or less, depending on how many aliases are contained in your copy of `default_host`. Also, the layout of the aliases shown above -- all nicely lined up one over the next -- was artificially produced. In reality the aliases were strung together in one long line.

Using the show and showAttribute methods to display contents of host alias

We've drilled down to the point where we've displayed a list of the aliases contained within the `default_host` virtual host. Now let's see how we can show the contents of one of those aliases.

Using show method to display contents

In the last exercise you displayed the config IDs of all the aliases. We're going to now use the config ID of *one of those aliases* to demonstrate how the `$AdminConfig` object can be used to extract the value of that alias.

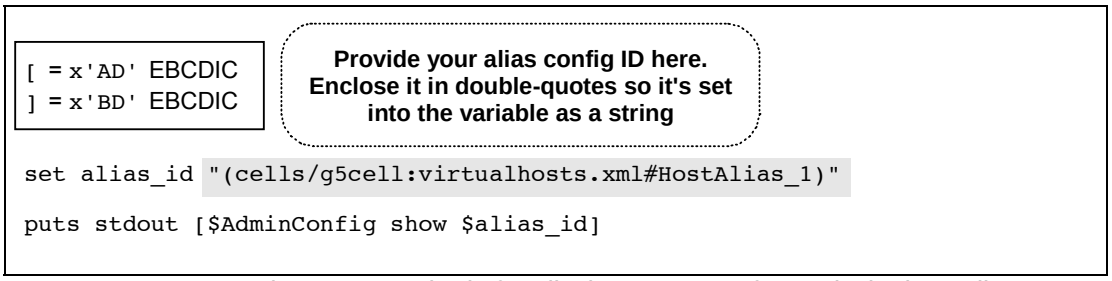
☐ Pick one of the aliases listed in the previous exercise and write the config ID here:

_____

Note: The config ID starts and ends with parenthesis -- `(` and `)` -- you may, if you wish, simply copy the config ID to the clipboard and then paste it into the Jacl script.

☐ *Copy* the file `/u/user1/lesson5e.jacl` to file `/u/user1/lesson5f.jacl`

☐ Edit the file `lesson5f.jacl` and add provide the following:



lesson5f.jacl - *Using `show` method of to display contents of a particular host alias*

- ☐ Issue the following command (all on one line):

```
./wsadmin.sh -javaoption -Dscript.encoding=Cp1047
  -conntype SOAP -host <host> -port <port> -f /u/user1/lesson5f.jacl
```

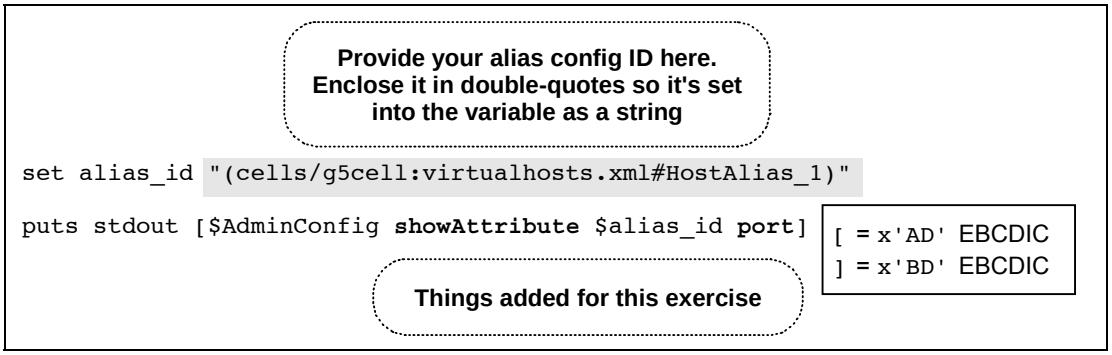
You should get back something that looks like this:

```
{hostname *}
{port 15518}
```

??? What this is saying is this: two attributes exist in this alias: `hostname` and `port`. The value for `hostname` is a single asterisk and the value for `port` is 15518.

Using showAttributes to display hostname attribute or port attribute individually

- ☐ *Copy* the file `/u/user1/lesson5f.jacl` to file `/u/user1/lesson5g.jacl`
- ☐ Edit the file `lesson5g.jacl` and add provide the following:



lesson5g.jacl - *Using `showAttribute` method to display value of port attribute*

- ☐ Issue the following command (all on one line):

```
./wsadmin.sh -javaoption -Dscript.encoding=Cp1047
  -conntype SOAP -host <host> -port <port> -f /u/user1/lesson5g.jacl
```

You should get back something that looks like this:

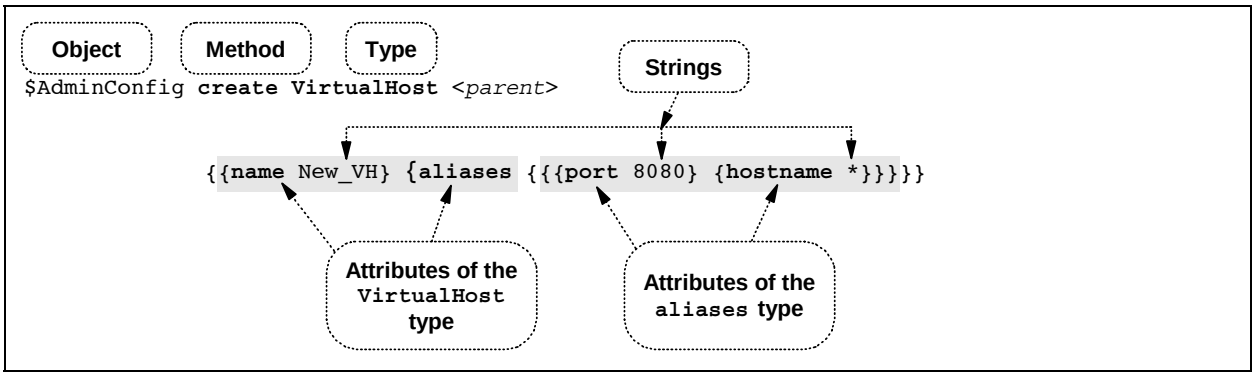
```
15518
```

??? That is just the value of `port` in a particular alias inside the `default_host` virtual host.

Creating a new virtual host complete with an HostName/Port alias

We're now ready to construct a Jacl script that will add a new virtual host to an environment, and populate that with an alias, complete with a `hostname` attribute and a `port` attribute.

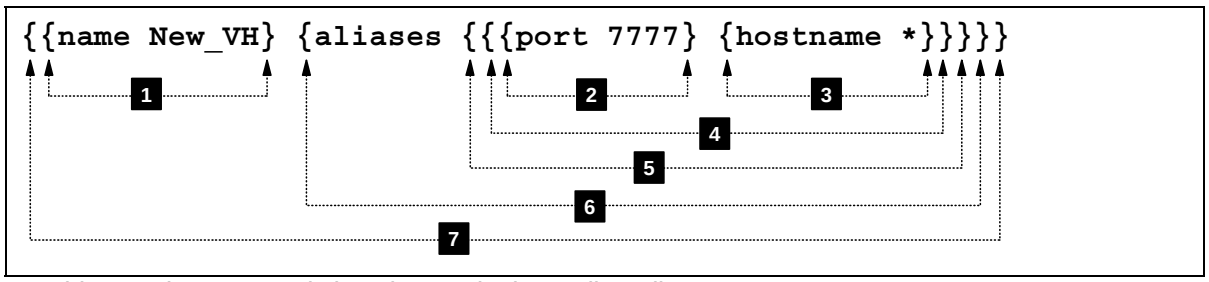
Earlier we showed this picture of what the create method command would look like to create a virtual host:



Revisit command to create a virtual host with alias

Understanding all the open and closing braces in create VirtualHost command

It's worth the time to pause and understand the braces in the attribute list for the command shown in the previous diagram.



Matching up the open and close braces in the attribute list

1 Encloses the `name` attribute, one of three attributes for the `VirtualHost` type

2 Encloses the `port` value for this particular alias

3 Encloses the `hostname` value for this particular alias

4 Marks the start and end of *this* alias, which contains `port 7777` and `hostname *`

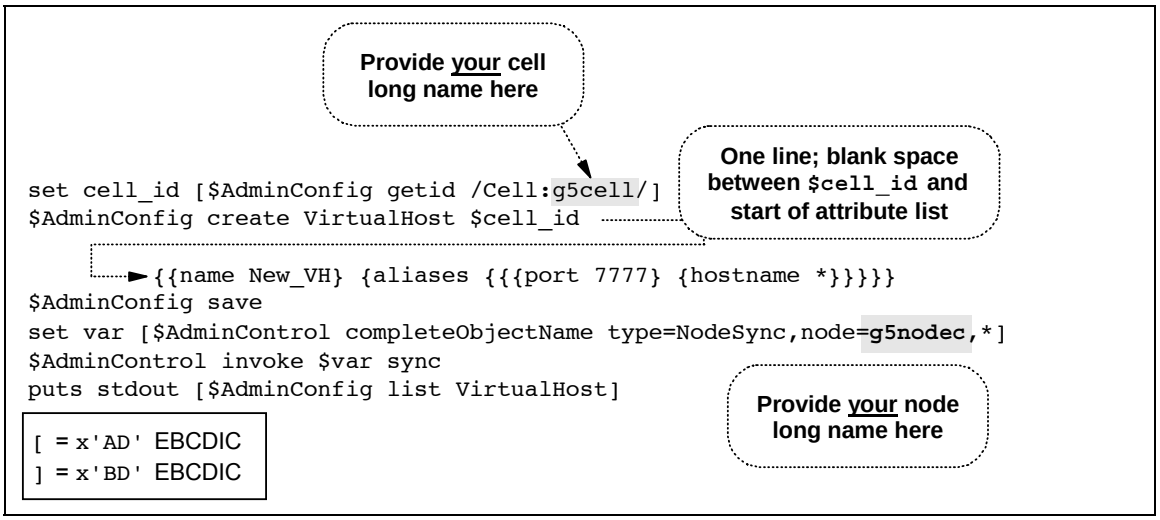
5 Marks the start and end of *all* the aliases that will be added to this new virtual host. This example shows only *one* port/hostname pair, but in reality you may add as many as you like. If you had another port/hostname pair, it would be another `{{port}{hostname}}` string *inside* the #5 braces but following the #4 braces.

6 Encloses the aliases keyword attribute and its attribute list.

7 Encloses *all* the attributes that follow the `VirtualHost` type.

Jacl script with hard-coded attribute list

- ☐ Create a file called `/u/user1/lesson5h.jacl`
- ☐ Code the following in the file:



lesson5h.jacl - *Jacl script to create new virtual host with one alias*

☐ Issue the following command (all on one line):

```
./wsadmin.sh -javaoption -Dscript.encoding=Cp1047
   -conntype SOAP -host <host> -port <port> -f /u/user1/lesson5h.jacl
```

You should get back a listing of the virtual hosts that are in your configuration repository. One of those virtual hosts should be `New_VH` created in this exercise.

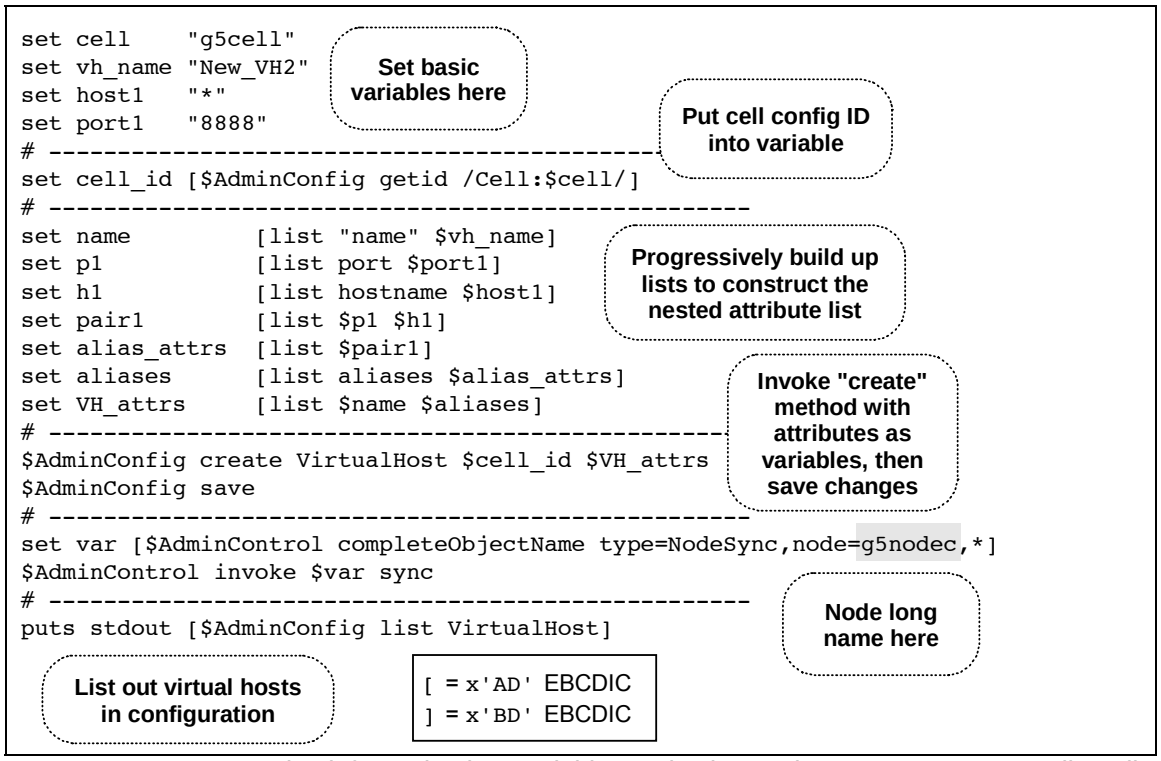
Jacl script using variables to populate attribute list

You could use the `lesson5h.jacl` script for another virtual host by simply changing some of the values. Of your could code the script so the command string was populated with variables.

Note: Or passed in as parameters. Here we'll show the variables being set at the top of the script. See "Passing values in as parameters" on page 27 for an example of how parameters can be passed in and parsed into variables.

☐ Create a file called `/u/user1/lesson5i.jacl`

☐ Code the following in the file:



lesson5i.jacl - *Putting information into variables and using Jacl* `list` *to construct attribute list*

Note: If the use of the Jacl `list` function as shown above is confusing, do the following:

- Understand that the variable set with this `list` function has an implied set of braces around it -- `set x [list a b c]` would result in `x` being equal to `{a b c}`
- Print out a copy of the picture shown under "Understanding all the open and closing braces in create VirtualHost command" on page 62
- Walk through the section of the code where all the list functions are used, and watch how the braces are built up, one after another.

☐ Issue the following command (all on one line):

```
./wsadmin.sh -javaoption -Dscript.encoding=Cp1047
   -conntype SOAP -host <host> -port <port> -f /u/user1/lesson5i.jacl
```

You should get back a listing of the virtual hosts in your environment.

Question: what if virtual host had multiple hostname/port pairs?

The example provided in `lesson5i.jacl` could easily be expanded to handle multiple aliases:

Note: This example is offered as `lesson5i-2.jacl`

```
set cell    "g5cell"
set vh_name "New_VH3"
set host1   "*"
set port1   "8888"
set host2   "www.myhost.com"
set port2   "9999"
# --------------------------------------------------
set cell_id [$AdminConfig getid /Cell:$cell/]
# --------------------------------------------------
set name         [list "name" $vh_name]
set p1           [list port $port1]
set h1           [list hostname $host1]
set pair1        [list $p1 $h1]
set p2           [list port $port2]
set h2           [list hostname $host2]
set pair2        [list $p2 $h2]
set alias_attrs  [list $pair1 $pair2]
set aliases      [list aliases $alias_attrs]
set VH_attrs     [list $name $aliases]
# --------------------------------------------------
$AdminConfig create VirtualHost $cell_id $VH_attrs
$AdminConfig save
# --------------------------------------------------
set var [$AdminControl completeObjectName type=NodeSync,node=g5nodec,*]
$AdminControl invoke $var sync
# --------------------------------------------------
puts stdout [$AdminConfig list VirtualHost]
```

Gray highlighting indicates added script

`[` = x'AD' EBCDIC
`]` = x'BD' EBCDIC

Modifying lesson5i.jacl so new virtual host had multiple aliases

??? Would it be possible to create a Jacl script that was even more dynamic than this? Yes, a looping structure could be built. But that won't be illustrated in this document. Here the objective was demonstrating how to build up nested lists to accomplish the goal of creating the new virtual host.

Using the modify method of `$AdminConfig` to add another alias to the virtual host

At this point you should have several new virtual hosts in your environment. Now what we'll do is use the modify method to change some aspect of one of those virtual hosts.

If you use the help `method` of `$AdminConfig` to see what it says about the method `modify`, it tells you the method takes two arguments: the config ID and "attributes." Here again we find ourselves relying on what we've learned elsewhere to know what "attributes" means in this context.

From earlier exercises we know the attributes of the VirtualHost configuration object: `name`, `aliases` (under which two sub-attributes exist: `hostname` and `port`), and `mimeTypes` (under which exists: `extensions` and `type`).

We'll start by modifying the `name` attribute of one of your virtual hosts:

Changing the name of the virtual host

- ☐ Create a file called `/u/user1/lesson5j.jacl`
- ☐ Code the following in the file:

```
set exist_name "New_VH2"
set new_name   "Mod_VH2"
# ------------------------------------------------------------
set vh_id   [$AdminConfig getid /VirtualHost:$exist_name/]
# ------------------------------------------------------------
set name_list  [list name $new_name]
set attr_list  [list $name_list]
# ------------------------------------------------------------
$AdminConfig modify $vh_id $attr_list
$AdminConfig save
# ------------------------------------------------------------
set var [$AdminControl completeObjectName type=NodeSync,node=g5nodec,*]
$AdminControl invoke $var sync
# ------------------------------------------------------------
puts stdout [$AdminConfig list VirtualHost]
```

[= x'AD' EBCDIC
] = x'BD' EBCDIC

Construction of the {{name Mod_VH2}} attribute list to `modify` method

`lesson5j.jacl` - *Jacl script that changes virtual host name from "exist_name" to "new_name"*

☐ Issue the following command (all on one line):

```
./wsadmin.sh -javaoption -Dscript.encoding=Cp1047
   -conntype SOAP -host <host> -port <port> -f /u/user1/lesson5j.jacl
```

You should get back a listing of the virtual hosts in your environment, including the modified one.

??? What happened there? The command passed in was:

```
$AdminConfig modify <config ID of VH> {{name Mod_VH2}}
```

It read the value name as an attribute type and simply replaced the same attribute for the object with the config ID given -- name in this case -- with the one supplied on the attribute list.

Adding additional aliases to the virtual host

It's a very similar process to that illustrated in the previous exercise. Rather than pass in the attribute list `{{name Mod_VH2}}`, we'll pass in:

```
{{aliases {{{port 5555} {hostname *}} {{port 6666} {hostname *}}}}}
```

Do the following:

☐ Create a file called `/u/user1/lesson5k.jacl`

☐ Code the following in the file:

```
set vh_name     "New_VH2"
set port1       "5555"
set host1       "*"
set port2       "6666"
set host2       "*"
# ------------------------------------------------------
set vh_id   [$AdminConfig getid /VirtualHost:$vh_name/]
# ------------------------------------------------------
set p1          [list port $port1]
set h1          [list hostname $host1]
set pair1       [list $p1 $h1]
set p2          [list port $port2]
set h2          [list hostname $host2]
set pair2       [list $p2 $h2]
set pair_list   [list $pair1 $pair2]
set alias_list [list aliases $pair_list]
set attr_list   [list $alias_list]
# ------------------------------------------------------
$AdminConfig modify $vh_id $attr_list
$AdminConfig save
# ------------------------------------------------------
set var [$AdminControl completeObjectName type=NodeSync,node=g5nodec,*]
$AdminControl invoke $var sync
# ------------------------------------------------------
puts stdout [$AdminConfig list VirtualHost]
```

[= x'AD' EBCDIC
] = x'BD' EBCDIC

Construction of the attribute list to `modify` method

`lesson5k.jacl` - *Jacl script that adds two more hostname/port aliases to the virtual host*

> **Note:** Notice how the `list` function is used to build the attribute lists from the *inside* working out? That's the easiest way to keep track of all the open and close braces.

☐ Issue the following command (all on one line):

`./wsadmin.sh -javaoption -Dscript.encoding=Cp1047`

`-conntype SOAP -host <`*`host`*`> -port <`*`port`*`> -f /u/user1/lesson5k.jacl`

You should get back a listing of the virtual hosts in your environment, including the modified one. Go into the Admin Console to look at the aliases added to that virtual host.

??? Could WSADMIN be used to list out the aliases? Sure. But we've already done that, so going into the Admin Console is simply easier at this point.

Deleting the test virtual hosts created in this lesson

To delete a virtual host requires the use of the `remove` method of `$AdminConfig`. The method has a very simple syntax:

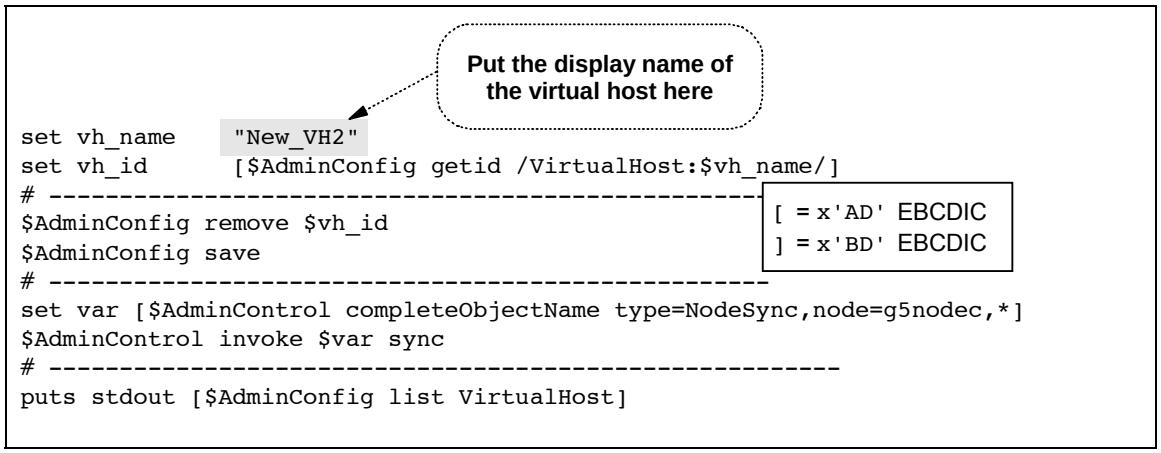
`$AdminConfig remove <`*`config ID of object being removed`*`>`

> **Note:** You can get the config ID for any object in the repository by using the `$AdminConfig getid` method. For a virtual host, the command would look like this:
>
> `$AdminConfig getid /VirtualHost:`*`VH_name`*`/`
>
> The resulting config ID will look *something* like this:
>
> `New_VH(cells/g5cell:virtualhosts.xml#VirtualHost_1079548227779)`

Jacl script to delete a virtual host

Do the following:

- ☐ Create a file called `/u/user1/lesson5l.jacl`
- ☐ Code the following in the file:




```
set vh_name    "New_VH2"
set vh_id      [$AdminConfig getid /VirtualHost:$vh_name/]
# ------------------------------------------------------
$AdminConfig remove $vh_id
$AdminConfig save
# ------------------------------------------------------
set var [$AdminControl completeObjectName type=NodeSync,node=g5nodec,*]
$AdminControl invoke $var sync
# ------------------------------------------------------
puts stdout [$AdminConfig list VirtualHost]
```

`lesson5l.jacl` - *Jacl script that deletes virtual host from configuration*

- ☐ Issue the following command (all on one line):

  `./wsadmin.sh -javaoption -Dscript.encoding=Cp1047`

  `-conntype SOAP -host <`*`host`*`> -port <`*`port`*`> -f /u/user1/lesson5l.jacl`

  The specified virtual host will be deleted and the remaining virtual hosts will be displayed.

Question: is it possible to delete multiple objects with one command invocation

Apparently not. It looks as though the remove method is capable of processing only one argument per invocation. If you pass two or more config IDs on the remove method, only the first is handled. The others are ignored.

Note: You could use the `lesson5l.jacl` script and simply copy the lines to form a bigger script that deletes the multiple objects. It would require multiple `$AdminConfig remove` lines in the script, but at least it would be an automated removal of multiple objects.

Creating a new server by copying from an existing server

In this exercise we will create a new application server by modeling it after an existing server. To do this we'll use the `createUsingTemplate` method. That method has the following arguments:

```
type, parent, attributes, template
```

where:

`type`	Is the type of configuration object being created. In this lesson it will be `Server`.
`parent`	Is the configuration object to which this new object will belong. Servers have nodes as their parents. This argument requires that the "config ID" of the node be given as the parent.
`attributes`	The attributes of the new server are provided here. For a server you must supply at a minimum the *long* name. The other attributes of the server will either default (such as *short* name: `BBOS001`) or be modeled after the template server's settings.
`template`	The "config ID" of the configuration object on which this new object will be modeled.

Note: WSADMIN can't be used to do any of the MVS-related things like create RACF profiles, copy JCL procedures, or create "static" WLM application environments. You can create a server in WSADMIN but you may have to do other things to get the server to actually start.

Simple example without node synchronization

- ☐ Create a file called `/u/user1/lesson5m.jacl`
- ☐ Code the following in the file:

```
[ = x'AD' EBCDIC
] = x'BD' EBCDIC

Put your existing server and node long names here

Put the new server's long name here

set model_serv   [$AdminConfig getid /Server:g5sr01c/]
set parent_node  [$AdminConfig getid /Node:g5nodec/]
$AdminConfig createUsingTemplate Server $parent_node {{name g5sr02c}} $model_serv
$AdminConfig save
```

`lesson5m.jacl` -- *simple creation of a new server, based on a model template*

??? The first two lines are simply creating variables into which the "config ID" of the template server and parent node will be held. The third line is what creates the server.

- ☐ Issue the following command (all on one line):

```
./wsadmin.sh -javaoption -Dscript.encoding=Cp1047
     -conntype SOAP -host <host> -port <port> -f /u/user1/lesson5m.jacl
```

You will see nothing back but the command prompt. If you go into the Admin Console and look under "Application Servers," you'll see this new server.

Note: Your new server is very rough at this point in time:

- It is not yet synchronized with the node; it is merely defined in the "Master Configuration" of the Deployment Manager
- The server's short name is set to the default value of `BB0S001`
- The server's "cluster transition name" is set to the default value of `BBOC001`
- The server's HTTP and End Point port values are defaulted

Message: what you did in the Jacl script was little more than what's accomplished by clicking the "New" button in the Admin Console to create a server.

- ☐ Go into the Admin Console and ***delete*** this newly created server. Be sure to save to the "Master Configuration".

??? We're about to make the script more powerful. We want that server removed from the configuration so we can re-use the same server name.

More automated example with node synchronization

We saw in "Initiating synchronization using WSADMIN" on page 50 how to invoke node synchronization. Let's now include that in a script to synchronize the change out to the node.

- ☐ Create a file called `/u/user1/lesson5n.jacl`
- ☐ Code the following in the file:

```
                Provide values here

set model_serv      "g5sr01c"
set new_serv_name   "g5sr02c"
set parent          "g5nodec"
# ------------------------------------------------------------
set m_ID       1    [$AdminConfig getid /Server:$model_serv/]
set p_ID            [$AdminConfig getid /Node:$parent/]
# ------------------------------------------------------------
set name_pair       [list name $new_serv_name]
set attr_list       [list $name_pair]                          2
# ------------------------------------------------------------
$AdminConfig createUsingTemplate Server $p_ID $attr_list $m_ID  3
$AdminConfig save
# ------------------------------------------------------------
set var [$AdminControl completeObjectName type=NodeSync,node=$parent,*]
$AdminControl invoke $var sync  4
```

[= x'AD' EBCDIC
] = x'BD' EBCDIC

lesson5n.jacl -- *more automated server creation with node synchronization*

1 Puts config IDs of model server and parent node into variables

2 Creates attribute list: `{{name g5sr02c}}` (in this example)

3 Drives createUsingTemplate method and saves changes

4 Derives "complete name" of `NodeSync` mBean in the parent node, then drives synchronization of the changes to the node

Note: Your new server is *still* a bit rough at this point in time: the short name, cluster transition name and port values are defaulting.

Changing an application server's short name

An application server's short name value can be changed using WSADMIN, but you have to understand where that value resides in the configuration heirarchy. In this exercise we'll show you how to determine where it is, then show you how to change it.

☐ Create a file called `/u/user1/lesson5o.jacl`

☐ Code the following in the file (see "note" regarding supplied sample file name):

```
                                  Put the server long
                                      name here

set server_long  "g5sr02c"
set server_id    [$AdminConfig getid /Server:$server_long/]
# ------------------------------------------------------
puts stdout [$AdminConfig show $server_id]
```

[= x'AD' EBCDIC
] = x'BD' EBCDIC

lesson5o.jacl - *Using the `show` method to display the configuration attributes*

Note: Sample file supplied as `lesson5o-pre.jacl`. We'll be modifying `lesson5o.jacl` in a moment, and the modified version is supplied as `lesson5o.jacl`.

- ☐ Issue the following command (all on one line):

./wsadmin.sh -javaoption -Dscript.encoding=Cp1047

-conntype SOAP -host <*host*> -port <*port*> -f /u/user1/lesson5o.jacl

What you'll see is *something* like this:

```
{components
 {(cells/g5cell/nodes/g5nodec/servers/g5sr02c:server.xml#NameServer_1080661509735)
  (cells/g5cell/nodes/g5nodec/servers/g5sr02c:server.xml#ApplicationServer_1080661509735)}}
{customServices {}}
{name g5sr02c}
{processDefinitions
 {(cells/g5cell/nodes/g5nodec/servers/g5sr02c:server.xml#JavaProcessDef_1080661509737)
  (cells/g5cell/nodes/g5nodec/servers/g5sr02c:server.xml#JavaProcessDef_1080661509738)}}
{serverInstance
 (cells/g5cell/nodes/g5nodec/servers/g5sr02c:server.xml#ServerInstance_1080661509737)}
{serverType APPLICATION_SERVER}
{services
 {(cells/g5cell/nodes/g5nodec/servers/g5sr02c:server.xml#PMIService_1080661509722)
  (cells/g5cell/nodes/g5nodec/servers/g5sr02c:server.xml#AdminService_1080661509722)
  (cells/g5cell/nodes/g5nodec/servers/g5sr02c:server.xml#ObjectRequestBroker_1080661509722)
  (cells/g5cell/nodes/g5nodec/servers/g5sr02c:server.xml#TraceService_1080661509725)}}
{shortName BBOS001}
{stateManagement
 (cells/g5cell/nodes/g5nodec/servers/g5sr02c:server.xml#StateManageable_1080661509722)}
{statisticsProvider
 (cells/g5cell/nodes/g5nodec/servers/g5sr02c:server.xml#StatisticsProvider_1080661509722)}
{uniqueId BAFEB6625BB8E1DE000002740000000109521847}
```

Output of the show command against an application server

Note: The layout of the actual output won't look as lined-up and neat as what's shown above. That was modified to help illustrate the different configuration attributes under the server.

From this we know that to change the server's short name, we need to modify the `shortName` attribute.

- ☐ Now *modify* the `lesson5o.jacl` file and include the following:

```
set server_long  "g5sr02c"
set new_short    "G5SR02C"
set parent       "g5nodec"
set server_id    [$AdminConfig getid /Server:$server_long/]
# ------------------------------------------------
set name_list    [list shortName $new_short]
set attr_list    [list $name_list]
# ------------------------------------------------------
$AdminConfig modify $server_id $attr_list
$AdminConfig save
# ------------------------------------------------------
set var [$AdminControl completeObjectName type=NodeSync,node=$parent,*]
$AdminControl invoke $var sync
```

[= x'AD' EBCDIC
] = x'BD' EBCDIC

lesson5o.jacl - *(modified) - changes server short name*

- ☐ Issue the following command (all on one line):

./wsadmin.sh -javaoption -Dscript.encoding=Cp1047

-conntype SOAP -host <*host*> -port <*port*> -f /u/user1/lesson5o.jacl

If successful, you should simply receive back the command prompt. To validate that the change was made, go into the Admin Console and display the "General Properties" of this server.

Changing an application server's Cluster Transition Name

This is a slightly more complicated task. A server's Cluster Transition Name is a property under the `ApplicationServer` type. In order to modify that property, we must first secure the config ID of that property. The process of drilling down to that config ID is a bit long:

- Get the config ID of the server
- Get the config ID of the `ApplicationServer` type under the server (there is only one)
- Get a listing of the config IDs of all the properties under the `ApplicationServer` type (there are several properties there, one of which is `ClusterTransitionName`)
- Parse out the first property, which by default should be the `ClusterTransitionName` property

☐ Create a file called `/u/user1/lesson5p.jacl`

☐ Code the following in the file:

```
set server_long   "g5sr02c"
set parent        "g5nodec"
set new_CTN       "G5SR02"
# ------------------------------------------------------------
set server_ID     [$AdminConfig getid /Server:$server_long/]
set appl_serv_ID  [$AdminConfig list ApplicationServer $server_ID]
set prop_list     [$AdminConfig list Property $appl_serv_ID]
set ctn_prop      [lindex $prop_list 0]
set first_seven   [string range $ctn_prop 0 6]
# ------------------------------------------------------------
if { !($first_seven=="Cluster") } then {
 puts stdout "Got wrong property!"
 exit
}
# ------------------------------------------------------------
set value_pair    [list value $new_CTN]
set attr_list     [list $value_pair]
$AdminConfig modify $ctn_prop $attr_list
$AdminConfig save
# ------------------------------------------------------------
set var [$AdminControl completeObjectName type=NodeSync,node=$parent,*]
$AdminControl invoke $var sync
```

1 Set these values manually

2

3

4

5

[= x'AD' EBCDIC
] = x'BD' EBCDIC

`lesson5p.jacl` - *Changing a server's ClusterTransitionName property*

1. Set variables specifying the server's long name, the parent node long name and the desired new "Cluster Transition Name" (CTN).
2. Drill down, capturing config IDs: first the server's, then the `ApplicationServer` type under the server, then get a listing of the `Property` types under the `ApplicationServer`. That list will contain more than one entry, so we use the Jacl `lindex` function to extract out the first. (Jacl lists are zero-offset, so the first item in the list is referenced with a 0.) Then we sub-string out the first seven character so we can sanity-check and make sure we got the right property. (We can't necessarily assume `ClusterTransitionName` will always be the first property.)
3. We check to see if the first seven characters of the property name is *not* equal to "Cluster". If it's not, then we got the wrong property and we exit. (Note: it would be possible to loop through the list and find the `ClusterTransitionName` property. That's simply more advanced Jacl. Our objective here is to show the fundamentals.)
4. We build the attribute list: `{{value G5SR02}}`, then modify the property based on the config ID of the property derived earlier.
5. Lastly, we synchronize the changes to the node.

- ☐ Issue the following command (all on one line):

```
./wsadmin.sh -javaoption -Dscript.encoding=Cp1047
      -conntype SOAP -host <host> -port <port> -f /u/user1/lesson5p.jacl
```

Concluding points on this lesson

As we stated at the beginning of this lesson, the use of `$AdminConfig` may not be of much value when the operation undertaken is a once-only thing. The time spent coding up the Jacl script might better be spent simply going through the Admin Console and creating the configuration object there.

But there are some `$AdminConfig` methods you *will* want to take advantage of: the `save` method and the mechanism shown earlier to synchronize changes with the nodes.

Perhaps the most important lesson derived from these exercises is the method you use to find *where things are kept in the configuration heirarchy*. We showed how the `show` method and the `showAttribute` method can be particularly useful for this purpose. And we discovered how critical the `getid` method is, because that's then used to make changes to specific objects in the configuration.

Lesson 6: The $AdminControl Object

The `$AdminControl` object is used to do things like start and stop servers, invoke node synchronization, and start/stop applications.

Note: Unlike `$AdminApp` and `$AdminConfig`, with `$AdminControl` you ***must*** be connected to a server process. If you use `-conntype none` and attempt to use `$AdminControl`, it'll tell you it can't process your request.

Does requiring a connection to server process limit how I might invoke `$AdminControl`?

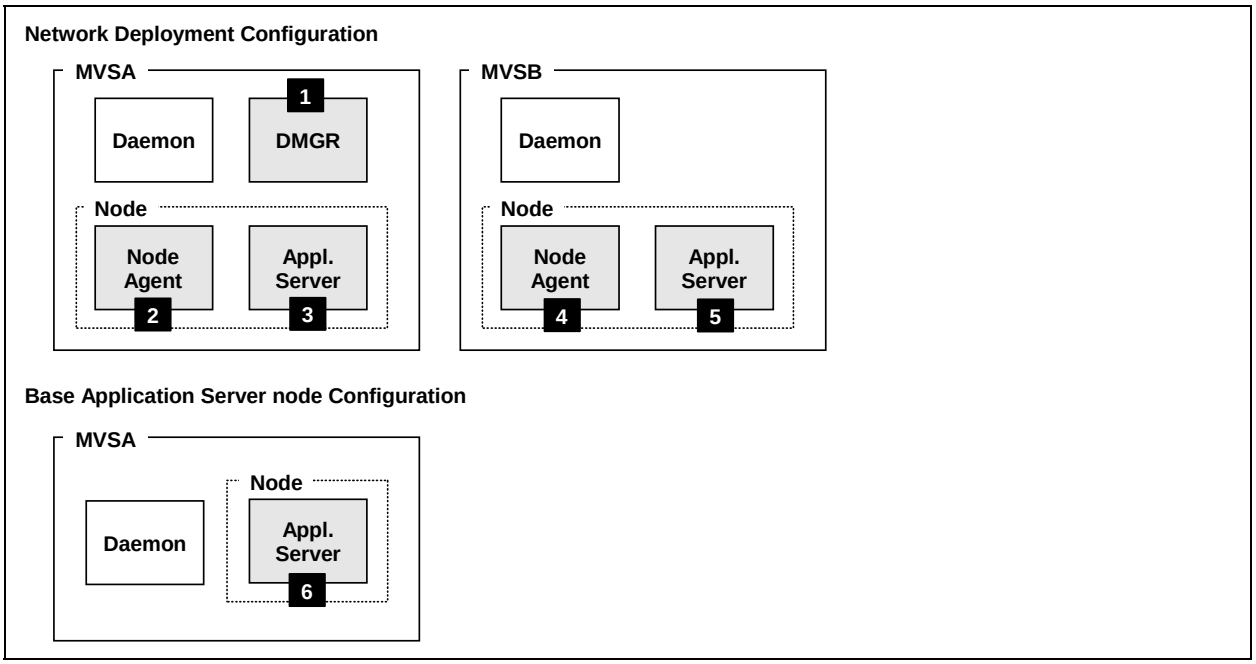
Just because `$AdminControl` requires that you be connected to a server process, you still have available to you all the ways to invoke the object:

- Starting WSADMIN with a connection to the server process, then issuing the command directly on the `wsadmin>` command prompt. This is great for simply commands, like that used to start a server.
- Invoking WSADMIN with a connection to the server process and passing in a pointer to a Jacl script in a file. Better for more complex commands that require capturing a config ID into a variable and then using that variable in another command.
- Submitting a batch JCL job that invokes `BPXBATCH`, which then invokes WSADMIN with a connection to a server process. The command may then be "inline" in the JCL, with the `-c` switch used to tell WSADMIN the commands follow, or in a separate file and pointed to with the `-f` switch. You may wish to use this when you don't have access to a OMVS or Telnet prompt.

Message: This is really no different from what we've been doing already. We connected to the Deployment Manager's server process to use the `$AdminConfig` object in the previous lesson. The only difference is that with `$AdminConfig` you had a choice; with `$AdminControl` you must connect to the server process.

Which server process should we connect to?

The following picture illustrates the options. Notes follow:



Different server processes available for connection

1 The Deployment Manager. Generally speaking, this is the recommended connection point, particularly if the Deployment Manager is up. It has the understanding of the complete cell, and is capable of executing the command to any point in the configuration. All `$AdminControl` methods are available when connected to the Deployment Manager.

2 The Node Agent for a particular node. Not all the `$AdminControl` methods will be available here. Control is limited to only the node in which the Node Agent resides.

3 An individual application server. Even fewer `$AdminControl` methods are available.

4 A different node's Node Agent. See note #2.

5 Another individual application server. See note #3.

6 In a Base Application Server node configuration, the only server process available is an application server. (Daemons do not have SOAP or RMI ports to which WSADMIN might connect; they are not eligible for WSADMIN connection.) `$AdminControl` methods are limited. For example, `stopServer` is available, but `startServer` is not. (You have to be connected to issue `$AdminControl` commands; if the server is not up you can't connect to it; if it's up you don't need to issue the `startServer` command; you can't start another server when connected to an application server process.)

Note: General rule of thumb: if you have a Deployment Manager up and running, use it. For the rest of this lesson we'll assume a connection to the Deployment Manager process, unless a different server process is explicitly stated.

Starting a server in a Network Deployment configuration

We'll start with a relatively easy method: `startServer`. The syntax of

`$AdminControl startServer <server long> <node long> <wait time>`

Where:

server long	Is the server's long name, such as `g5sr01c`.
node long	Is the node long name in which that server resides. If you're connected to the Node Agent you do not need this because it's assumed the server is in that Node Agent's node. But the Deployment Manager needs that because it may have several nodes. An example of a node long name would be `g5nodec`.
wait time	The time, in milliseconds, WSADMIN will wait before issuing the start command. If you omit this operand the start command is issued immediately.

Do the following:

☐ From an OMVS or Telnet prompt, issue the command:

`./wsadmin.sh -conntype SOAP -host <host> -port <port>`

Where <*port*> is the SOAP port of your Deployment Manager server.

☐ Issue the command:

`$AdminControl startServer <server long> <node long>`

☐ Go over to MVS and watch the server come up.

Note: It takes some time for a controller and its servant to come up. WSADMIN will wait until it sees the server up. If you're coming in on a Telnet session, it may time out before this happens.

Stopping a server in a Network Deployment configuration

Now stop the very server you just started:

- ☐ Issue the command:

  `$AdminControl stopServer <server long> <node long>`

- ☐ Go over to MVS and watch the server come down.

> **Note:** Coming down is a much quicker process than coming up.

Starting a Network Deployment server using batch JCL

Just to illustrate how this would be done, do the following:

- ☐ Create a batch JCL job with the following content:

```
//************************************************
//*  STEP 1 - Start Server
//************************************************
//INST1 EXEC PGM=IKJEFT01,REGION=0M
//SYSTSPRT DD  SYSOUT=*
//SYSTSIN  DD  *
 BPXBATCH SH +
   /wasv5config/g5cell+
   /DeploymentManager+   1
   /bin/wsadmin.sh +
   -conntype SOAP +
2  -host wsc3.washington.ibm.com +
   -port 15510 +
   -c '$AdminControl startServer g5sr01c g5nodec' +   3
   1> /tmp/lesson6a.out +
   2> /tmp/lesson6a.err
/*
//************************************************
//*  STEP Copy - Copy script output back to joblog
//************************************************
//DIAPPC EXEC PGM=IKJEFT01,REGION=0M
//SYSEXEC  DD  DISP=SHR,DSN=WAS502.WAS.SBBOEXEC
//SYSTSIN  DD *
   BBOHFSWR '/tmp/lesson6a.out'
   BBOHFSWR '/tmp/lesson6a.err'
//SYSTSPRT DD SYSOUT=*
//
```

`lesson6a.jcl` - *Starting of server from batch JCL*

1. The directory path to the copy of `wsadmin.sh` being invoked. In truth, this does not have to be the Deployment Manager's directory. It could be any copy of `wsadmin.sh` -- even one on another platform. But if you use an off-platform copy and you have security turned on, then there needs to be a coordination of certificates. Easier to simply invoke `wsadmin.sh` on the MVS image.
2. The specification of the connection type, host and port. Make this the DMGR's SOAP port.
3. The command string as used before. Note the single quotes surrounding that command string.

> **Note:** Again, the time it takes for a server to come up may be longer than the default timeout periods. When we ran this, we saw a RC=12 even though the server did come up. The error message was: "Exception: Read timed out." But the server did come up.

Starting a server in a Base Application Server node configuration

Can't be done. For `startServer` to work, you must be connected to some server process. In a Base Application Server node there is only the application server. (The Daemon is not eligible for WSADMIN connection.) Even if you have two application servers defined within one Base Application Server node (something we do *not* recommend), issuing `startServer` in one will not result in the starting of another.

Note: This is true from the Admin Console as well. In a Base Application Server node, the Admin Console running in one application server has limited ability to affect other servers. It can install an application into the other server, but it can't start or stop the other server, and it can't start or stop applications in that other server. Further reason why we don't recommend more than one application server in a Base Application Server node configuration.

Stopping a server in a Base Application Server node configuration

This *is* possible. The `stopServer` method can be used in a Base Application Server node configuration to stop the server to which WSADMIN is connected. The command is simply:

```
$AdminControl stopServer <server long name>
```

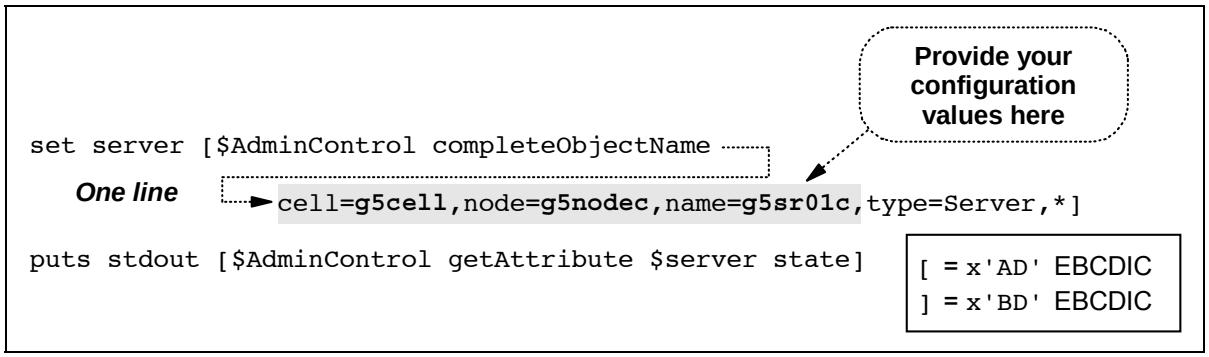
The server will come down rather quickly. If you invoked that command at an OMVS or Telnet prompt, you'll be returned to the prompt. But you'll no longer be connected to a running server process.

Checking the status of a server process

This is a two-step process: first the "complete object name" of the Mbean needs to be queried and put into a variable, then that variable is used in a second command to get the server's status.

Do the following:

- ☐ Create a file called `/u/user1/lesson6b.jacl`
- ☐ Code the following in the file:



`lesson6b.jacl` - *Getting the status of a server*

- ☐ Issue the following command (all on one line):

```
./wsadmin.sh -javaoption -Dscript.encoding=Cp1047
      -conntype SOAP -host <host> -port <port> -f /u/user1/lesson6b.jacl
```

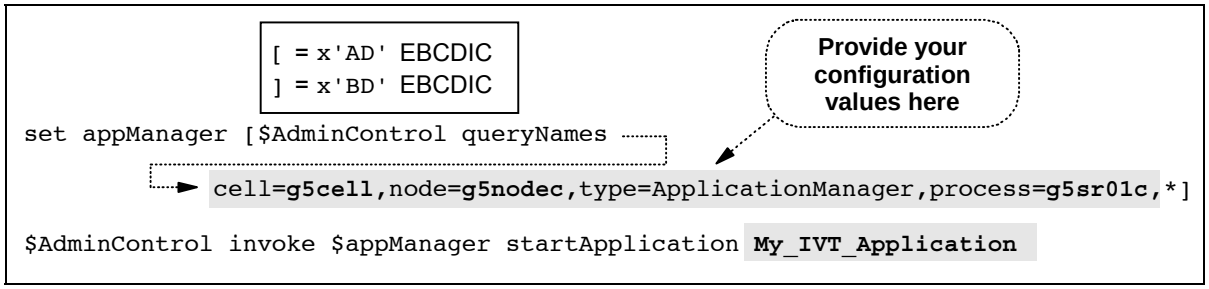
You'll get back the status of the named server: either `STARTED` or `STOPPED`.

Note: You could, of course, put the output from this into a variable, and then test against the variable to see whether issuing a `startServer` command is necessary. It all depends on how fancy you want your Jacl scripts to be.

Starting an Application

This is another two-step process: get the Mbean name, then drive one of the methods on the Mbean to start the application.

- ☐ Create a file called `/u/user1/lesson6c.jacl`
- ☐ Code the following in the file:



`lesson6c.jacl` - *Starting an application in a server*

- ☐ Issue the following command (all on one line):

```
./wsadmin.sh -javaoption -Dscript.encoding=Cp1047
      -conntype SOAP -host <host> -port <port> -f /u/user1/lesson6c.jacl
```

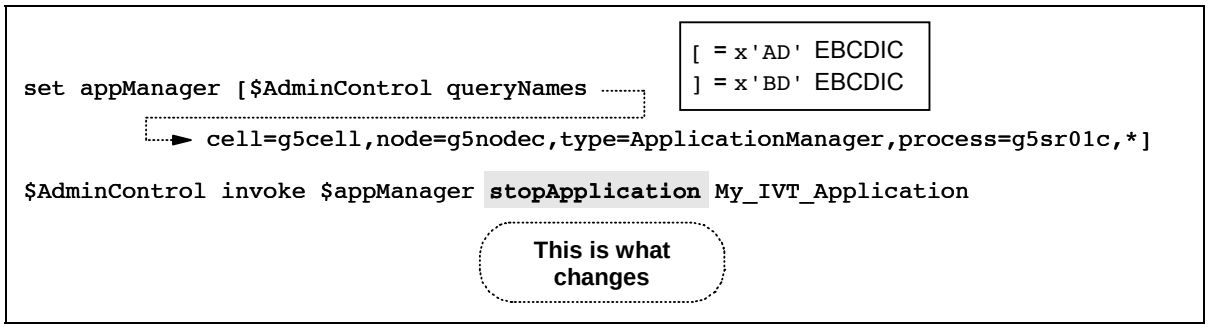
You don't get anything back; the application simply starts.

> **Note:** You could immediately check the status of the application to insure it was up. See "Checking the status of an application" on page 79.

Stopping an Application

This is more or less the reverse of the previous lesson.

- ☐ Copy the `lesson6c.jacl` file into a file called `/u/user1/lesson6d.jacl`
- ☐ Modify the file slightly:



`lesson6d.jacl` - *Stopping an application in a server*

- ☐ Issue the following command (all on one line):

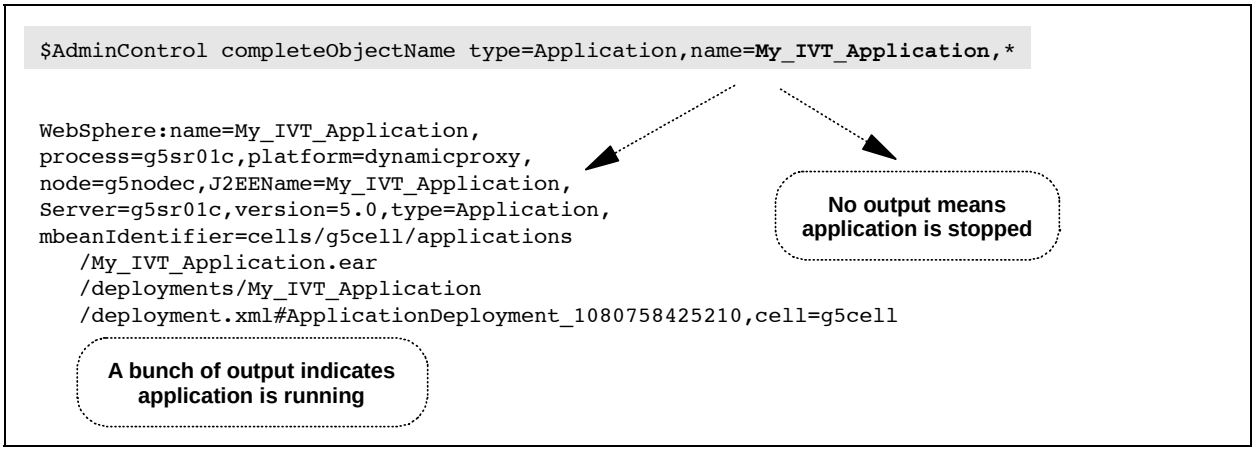
```
./wsadmin.sh -javaoption -Dscript.encoding=Cp1047
      -conntype SOAP -host <host> -port <port> -f /u/user1/lesson6d.jacl
```

Again, you don't get anything back; the application simply stops.

> **Note:** You could immediately check the status of the application to insure it was down. See "Checking the status of an application" on page 79.

Checking the status of an application

The process here is rather simple: attempt to get the "complete object name" of the application. If the `$AdminControl` object returns a value, the application is started. But if `$AdminControl` returns null, then the application is down:



Checking the status of an application using `completeObjectName` method

We can use this knowledge and test the length of the output. Do the following:

- ☐ Create a file called `/u/user1/lesson6e.jacl`
- ☐ Code the following into the file:

```
set app_name  "My_IVT_Application"
# ---------------------------------------------
set status    [$AdminControl completeObjectName type=Application,name=$app_name,*]
# ---------------------------------------------
set length    [string length $status]
if { ($length==0) } then {
 puts stdout "Application is STOPPED"
} else {
 puts stdout "Application is STARTED"
}
```

[= x'AD' EBCDIC
] = x'BD' EBCDIC

`lesson6e.jacl` - *Simple checking of length of output to test for application start/stop status*

- ☐ Issue the following command (all on one line):

```
./wsadmin.sh -javaoption -Dscript.encoding=Cp1047
      -conntype SOAP -host <host> -port <port> -f /u/user1/lesson6e.jacl
```

And the output will be either `STOPPED` or `STARTED` based on the status of the application.

Concluding points on this lesson

The `$AdminControl` object may prove quite useful when developers and testers don't have access to the MVS console or the Admin Console, but wish to do simple things like start and stop servers, or start and stop applications. Things like the creation of servers will *probably* be done outside of WSADMIN simply because MVS requires other things -- RACF profiles, JCL procedures -- along with the server definition inside of WebSphere.

Lesson 7: Digging Deeper into the $AdminApp Object

Back in "Lesson 4: Installing an Application using $AdminApp Object" on page 35 we explored the `$AdminApp` object to do some very simple application installations. Here we'll expand on that and do a bit more.

Please Read: At this point we'll stop offering some of the explicit instructions on the creation of Jacl script files and the invocation of WSADMIN to process the script.

Installing a second copy of an application into a cell

WebSphere permits the same application to be installed multiple times into the same cell. The only requirement is that the "appname" be unique. The appname for the `MyIVT.ear` application is `My_IVT_Application`, a value that's defined in the `application.xml` file inside the EAR. You may override that value when installing an application:

```
$AdminApp install /u/user1/MyIVT.ear
   {-node g5nodec -server g5sr01c -appname My_IVT_Application2}

$AdminConfig save
set var [$AdminControl completeObjectName type=NodeSync,node=g5nodec,*]
$AdminControl invoke $var sync
```

New appname to be used when installing this copy of the application

Changes synchronized out to node

`lesson7a.jacl` - *Providing a different "appname" from what's defined in EAR file*

Note: Any time an application is installed into a Network Deployment configuration through the Deployment Manager, be sure to synchronize to the nodes. See "Synchronizing changes with nodes" on page 49.

Setting the JNDI name for an EJB

The `$AdminApp install` option that is used to map a JNDI name to an EJB is:

```
BindJndiForEJBNonMessageBinding
```

Note: As the name implies, this is for an EJB that is *not* a message driven bean. If the EJB is an MDB, the option is `BindJndiForEJBMessageBinding`.

The `MyIVT.ear` sample application has within it a stateless session bean. We can use WSADMIN to determine if `BindJndiForEJBNonMessageBinding` applies to the EAR file by using the `options` method to query against the EAR:

```
$AdminApp options /u/user1/MyIVT.ear
```

What we receive back is a somewhat lengthy listing of the applicable options:

```
WASX7112I: The following tasks are valid for "/u/user1/MyIVT.ear"
BindJndiForEJBNonMessageBinding
MapEJBRefToEJB
MapWebModToVH
MapModulesToServers
EnsureMethodProtectionFor10EJB
preCompileJSPs
nopreCompileJSPs
```

The option related to the binding of the JNDI name to the stateless session bean

(lines removed to save space in the document)

```
defaultbinding.ejbjndi.prefix
defaultbinding.virtual.host
defaultbinding.force
defaultbinding.strategy.file
```

Output of `options` method for `MyIVT.ear`, and display of `BindJndiForEjbNonMessageBinding`

What's the syntax of the `BindJndiForEJBNonMessageBinding` option? Here the InfoCenter offers some help. A search on that option key word yielded:

BindJndiForEJBNonMessageBinding

Binds enterprise beans to Java Naming and Directory Interface (JNDI) names.
Ensure each non message-driven enterprise bean in your application or module is bound to a JNDI name. Use this option to provide missing data or update a task.

Example usage:

Using Jacl:

```
$AdminApp install myapp.ear {-BindJndiForEJBNonMessageBinding
{{"Increment Bean Jar" Inc Increment.jar,META-INF/ejb-jar.xml IncBean}}}
```

Use the **taskInfo** command of the AdminApp object to obtain information about the data needed for your application. You only need to provide data for rows or entries that are missing information, or those where you want to update the existing data.

InfoCenter listing for `BindJndiForEJBNonMessageBinding` option

As the InfoCenter write-up suggests, the `taskInfo` method of `$AdminApp` can be used to provide information about the present settings in the EAR file. Here's what comes back:

```
$AdminApp taskInfo /u/user1/MyIVT.ear BindJndiForEJBNonMessageBinding
  :
WASX7348I: Each element of the BindJndiForEJBNonMessageBinding task
consists of the following 4 fields:
  :
"EJBModule", "EJB", "uri", "JNDI".
  :
The current contents of the task after running default bindings are:
EJBModule: My IVT EJB Module Display Name
EJB: My_IVT_EJB_Name
uri: MyIVTStatelessSession.jar,META-INF/ejb-jar.xml
JNDI: ejb/My_IVT_Session_Bean_JNDI_Name
```

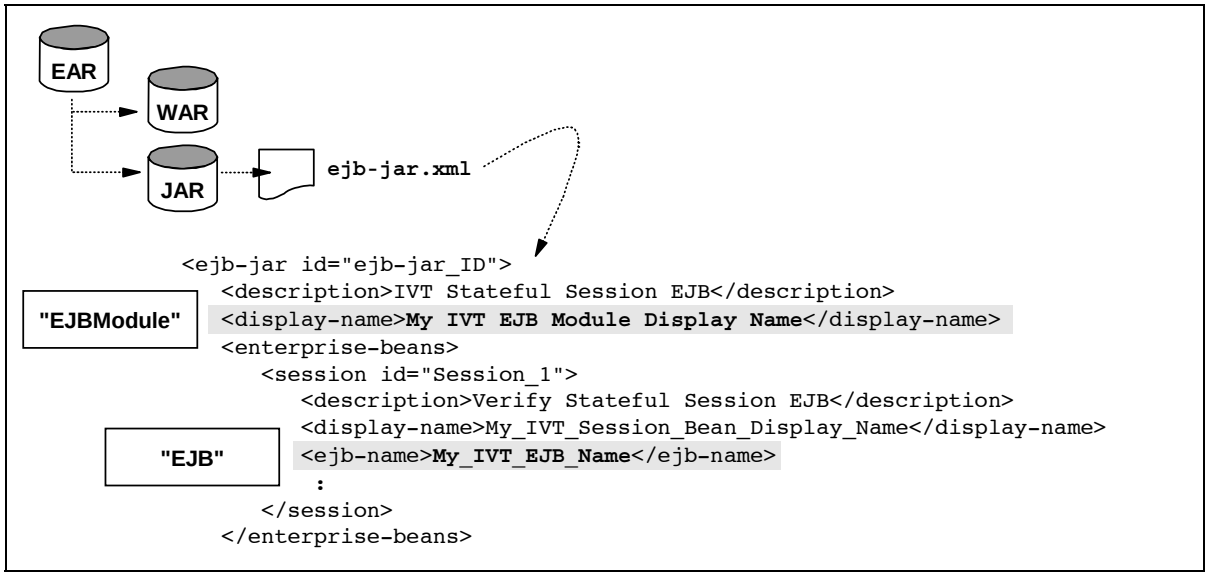
Contents of `MyIVT.ear` and its `BindJndiForEJBNonMessageBinding` value

Note: Only the `JNDI` value can be assigned a value. The other three are used to identify the EJB to which the JNDI name will be bound. Those must be set equal to the actual values of the bean for which the JNDI name is being provided.

Where are those values to be found in the EAR file itself?

Those four values -- `EJBModule`, `EJB`, `uri` and `JNDI` -- are all contained within the EAR file itself. That's why the `taskInfo` method is able to go directly against the EAR file and get the information.

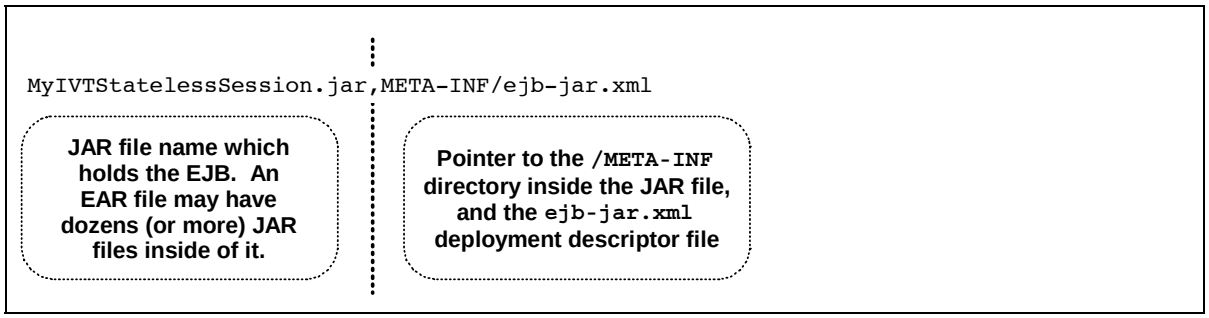
The first two values -- `EJBModule` and `EJB` -- are found in the `ejb-jar.xml` file, which is in EJB's JAR file, which is inside the EAR file:



Where `taskInfo` got EJBModule and EJB information

Note: You can see that the "EJBModule" value has blank spaces in its value. That's why that needs be coded inside double-quotes on the `BindJndiForEJBNonMessageBinding` option.

The `uri` value is simply a pointer to the JAR file found in the EAR file, with directions to the `ejb-jar.xml` file:



Value `uri` is pointer to the `ejb-jar.xml` file inside the JAR file

Note: If the `MyIVT.ear` file had more than one EJB inside it, the `taskInfo` function would provide more than one set of values for `BindJndiForEJBNonMessageBinding`. But `MyIVT.ear` has only one EJB, so it returned only one set of information.

Finally, the value for `JNDI` was found in the `ibm-ejb-jar-bnd.xmi` file:

```
<ejbBindings xmi:id="Session_1_Bnd" jndiName="ejb/My_IVT_Session_Bean_JNDI_Name">
```

This is the JNDI value set by the developer when the EAR file was exported from WSAD. This is what we're seeking to change during the installation of this application.

Constructing `$AdminApp` install command with change to JNDI name

Now we're prepared to construct the command to install `MyIVT.ear` with a different JNDI name:

```
$AdminApp install /u/user1/MyIVT.ear
{-node g5nodec -server g5sr01c
  -BindJndiForEJBNonMessageBinding
  {{"My IVT EJB Module Display Name"
     My_IVT_EJB_Name
     MyIVTStatelessSession.jar,META-INF/ejb-jar.xml
     ejb/New_JNDI}}}
set var [$AdminControl completeObjectName type=NodeSync,node=g5nodec,*]
$AdminControl invoke $var sync
```

Whole thing on one line

New JNDI name for the EJB identified with the first three parameters

`lesson7b.jacl` - *Installing `MyIVT.ear` with a different JNDI name for the EJB*

Important: This application has a servlet that references the EJB. The bindings inside the EAR should correctly map the servlet to the EJB. If you change the EJB's JNDI name without changing the servlet's mapping, the application will install okay but the application won't work. That's why the next topic -- mapping an EJB-ref to a JNDI name -- is provided.

Before Proceeding: Delete the application and synchronize the nodes.

Mapping an EJB-ref to JNDI name

We know from our running of the options method against the `MyIVT.ear` file that there's an mapping to the EJB:

```
WASX7112I: The following tasks are valid for "/u/user1/MyIVT.ear"
BindJndiForEJBNonMessageBinding
MapEJBRefToEJB
MapWebModToVH
 :  ( lines removed)
defaultbinding.strategy.file
```

It saw an `<ejb-ref>` tag in the `web.xml` file:

```
<ejb-ref id="EjbRef_1">
   <ejb-ref-name>ejb/ivtEJBObject</ejb-ref-name>
   <ejb-ref-type>Session</ejb-ref-type>
   <home>com.ibm.websphere.ivt.ivtEJB.ivtEJBHome</home>
   <remote>com.ibm.websphere.ivt.ivtEJB.ivtEJBObject</remote>
   <ejb-link>My_IVT_EJB_Name</ejb-link>
</ejb-ref>
```

This is what provides the connection to the EJB. This is the "EJB" value we saw in the last exercise

How the `options` method knew that `MapEJBRefToEJB` was a valid option

We can use the `taskInfo` method to see what the present values for `MapEJBRefToEJB` are for the `MyIVT.ear` application file:

```
$AdminApp taskInfo /u/user1/MyIVT.ear MapEJBRefToEJB
 :
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
referenceBinding: ejb/ivtEJBObject
class: com.ibm.websphere.ivt.ivtEJB.ivtEJBObject
JNDI: ejb/My_IVT_Session_Bean_JNDI_Name
```

Six values to be coded in the `MapEJBRefToEJB` option. First five need to match what's found in EAR file ... the last you may change as needed.

Using `taskInfo` to determine present value of `MapEJBRefToEJB` in `MyIVT.ear`

??? Where did it get all that information? Again, it's all buried in the deployment descriptors of the EAR file. Hence the value of `taskInfo` to extract it relatively easily.

The InfoCenter offers an example of how to construct an `$AdminApp install` command that used the `-MapEJBRefToEJB` option. It looked like this:

```
$AdminApp install /u/user1/MyIVT.ear
{-node g5nodec -server g5sr01c
 -MapEJBRefToEJB
  {{"My_IVT_Webapp_Display_Name"
1  ""
   MyIVTWebApp.war,WEB-INF/web.xml
   ejb/ivtEJBObject 2
   com.ibm.websphere.ivt.ivtEJB.ivtEJBObject
3  ejb/New_JNDI}}}
set var [$AdminControl completeObjectName type=NodeSync,node=g5nodec,*]
$AdminControl invoke $var sync
```

Whole thing on one line

New EJB reference. Here we're setting it to the same value we used earlier when we changed the JNDI name for the EJB itself.

`lesson7c.jacl` - *Installing `MyIVT.ear` with and changing the `MapEJBRefToEJB` value*

1. The double-quotes with nothing inside act as a place holder. This option expects and requires six values. They are position sensitive. When we ran determined the settings for `MapEJBRefToEJB` for `MyIVT.ear`, we saw that the option "EJB" had no value. So here we're holding the place with a null.
2. This is the value coded on the "java:comp" inside the web application. This is the "symbolic" reference to the EJB. What we're doing here is mapping this symbolic to the actual JNDI name.
3. If you want this application to actually *work* -- rather than simply install properly -- then make sure this value accurately maps to an actual JNDI name bound to the EJB you hope to drive.

Before Proceeding: Delete the application and synchronize the nodes.

Putting it all together -- install application, set JNDI name, map reference to EJB

Let's now construct a single Jacl script that's does multiple things. We're going to rely on the Jacl variables to hold all the information, then we'll construct the command to install the application into the server and provide a new JNDI name for the EJB and provide a remapping of the webapp so it points to the newly-named EJB.

```
# ------------------------------------------------------------------
# Application and Target settings
# ------------------------------------------------------------------
set ear_file         "/u/user1/MyIVT.ear"
set server           "g5sr01c"                  1
set node             "g5nodec"
# ------------------------------------------------------------------
# BindJndiForEJBNonMessageBinding settings
# ------------------------------------------------------------------
set b_option         "-BindJndiForEJBNonMessageBinding"
set b_mod            "\"My IVT EJB Module Display Name\""
set b_ejb       2    "My_IVT_EJB_Name"
set b_uri            "MyIVTStatelessSession.jar,META-INF/ejb-jar.xml"
set b_jndi           "ejb/New_JNDI"
# ------------------------------------------------------------------
# MapEJBRefToEJB settings
# ------------------------------------------------------------------
set m_option         "-MapEJBRefToEJB"
set m_mod            "\"My_IVT_Webapp_Display_Name\""
set m_ejb            "\"\""
set m_uri       3    "MyIVTWebApp.war,WEB-INF/web.xml"
set m_ref            "ejb/ivtEJBObject"
set m_class          "com.ibm.websphere.ivt.ivtEJB.ivtEJBObject"
set m_jndi           $b_jndi  4
# ------------------------------------------------------------------
# Construct BindJndiForEJBNonMessageBinding option list
# ------------------------------------------------------------------
set b_string "$b_mod $b_ejb $b_uri $b_jndi"
set b_list [list $b_string]                     5
# ------------------------------------------------------------------
# Construct MapEJBRefToEJB option list
# ------------------------------------------------------------------
set m_string "$m_mod $m_ejb $m_uri $m_ref $m_class $m_jndi"
set m_list [list $m_string]                                    6
# ------------------------------------------------------------------
# Construct $AdminApp command
# ------------------------------------------------------------------
set cmd_opt_list [list -node $node -server $server]
lappend cmd_opt_list $b_option
lappend cmd_opt_list $b_list      7
lappend cmd_opt_list $m_option
lappend cmd_opt_list $m_list
$AdminApp install $ear_file $cmd_opt_list
$AdminConfig save
# ------------------------------------------------------------------
# Invoke node synchronization
# ------------------------------------------------------------------ 8
set var [$AdminControl completeObjectName type=NodeSync,node=$node,*]
$AdminControl invoke $var sync
```

`lesson7d.jacl` - *Jacl script that installs, sets JNDI, maps reference*

1 The server and node long names are set, along with the EAR file path and name.

2 The JNDI values are set. These variables will be used later to construct what will be a very long command. Using variables here making changing the values for a different application much easier. Coding the values in the long command string makes modifying that string challenging. The `$b_mod` variable has backslashes to "force" the double-quotes into the actual variable itself.

3 The values used to map the servlet to the EJB are set here. The `$m_mod` variable is like the `$b_mod` variable -- it has double-quotes as part of the variable itself.

4 To insure the JNDI name bound to the EJB is the same value used in the servlet's reference, we use the JNDI name variable in the reference.

5 Here we're building the options list for the JNDI name binding. The `$b_mod` variable presents a challenge: it has double-quotes in its value. If `a="a"` and `b=b`, then `[list $a $b]` results in: `{{"a"} b}`. In other words, the variable with double-quotes is placed inside its own list. What we want is `{"a" b}`. We get around this by placing the values in a string variable first, then into a list.

6 The same thing as #5, except for the option list for the EJB reference.

7 Here we constructed the `$AdminApp` command itself. We needed to insure all the option lists were properly nested and the proper number of braces were present and balanced. The command we were seeking to construct was this (all on one line):

```
$AdminApp install /u/user1/MyIVT.ear {-node g5nodec -server g5sr01c
 -BindJndiForEJBNonMessageBinding {{"My IVT EJB Module Display Name"
 My_IVT_EJB_Name MyIVTStatelessSession.jar,META-INF/ejb-jar.xml
 ejb/New_JNDI}} -MapEJBRefToEJB {{"My_IVT_Webapp_Display_Name" ""
 MyIVTWebApp.war,WEB-INF/web.xml ejb/ivtEJBObject
 com.ibm.websphere.ivt.ivtEJB.ivtEJBObject ejb/New_JNDI}}}
```

Notice how there's an option list that follows the EAR file name, and then two sets of nested option lists inside: the `-BindJndiForEJBNonMessageBinding` list and the `-MapEJBRefToEJB` list. We used the `lappend` Jacl function to add items to an already existing list. This wasn't strictly necessary ... we could have simply built a really long line that went well beyond the right side of the screen. This illustrates how to keep everything within an 80-column format if that's what you desire.

8 The node is synchronized. The `$node` variable is used to insure the node into which the application was installed in the node synchronized.

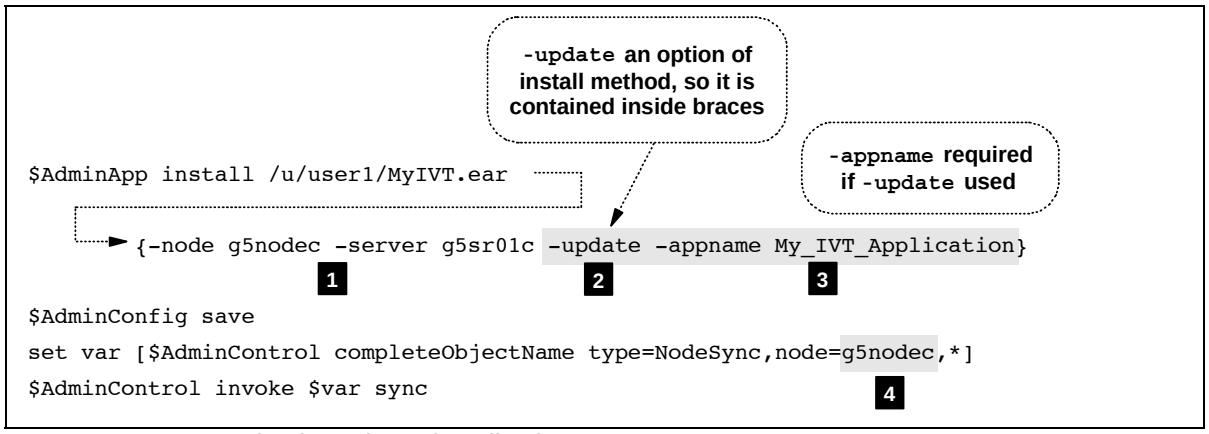
Hint: When developing a complex script like that it's good to do plenty of testing ahead of time. Rather than actually issuing the command when you first try your script, use `puts stdout` to echo back the command you built. That way you may inspect it visually to make sure you have all the appropriate braces and such.

Updating an existing application with a new copy

Suppose you have an application installed and you wish to update it with a new version of the application. You have two choices: you may manually uninstall and reinstall the application, or you may *update* the application.

Simple update

Here we used the `-update` option of the `install` method. The command looked like this:



`lesson7e.jacl` - *Simple update of application*

1 Be sure to specify `-node` and `-server`. If this is omitted, it will attempt to install the application into the server process to which WSADMIN is connected. In this example we're connected to the Deployment Manager. That *is* an application server, but not one in which any application other than the Admin Console should be running

2 `-update` is an option of the `install` method of `$AdminApp`. Therefore, it goes *inside* the braces like the other options.

3 The option `-appname` is required if `-update` is used. This tells WebSphere *what* application to update. You may have multiple copies of the `MyIVT.ear` application installed, even in the same application server (that's uncommon, but possible).

4 Be sure to specify the proper node on the synchronization command.

??? Does it matter if the application is started or not? No. If the application is started when the node synchronization takes place, the application will be stopped and restarted automatically.

Update of application and ignoring bindings in EAR file

The `-update` option will, by default, take whatever bindings are in the EAR file and replace the bindings found in the WebSphere configuration repository. Is it possible to maintain the bindings in the configuration repository and ignore the bindings in the EAR file? Yes. That's done with the `-update.ignore.new` option.

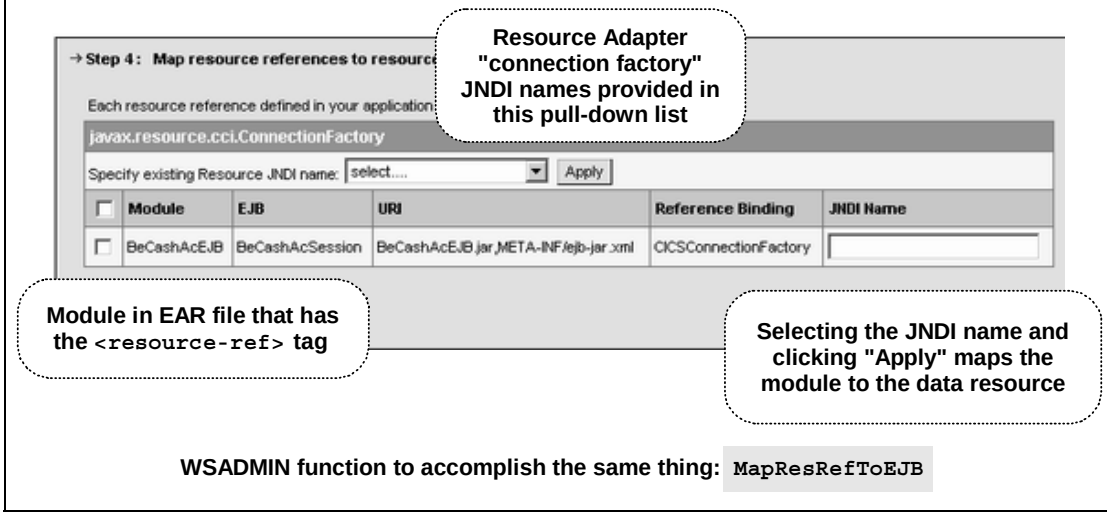
```
$AdminApp install /u/user1/MyIVT.ear
    {-node g5nodec -server g5sr01c
         -update -update.ignore.new -appname My_IVT_Application}
$AdminConfig save
set var [$AdminControl completeObjectName type=NodeSync,node=g5nodec,*]
$AdminControl invoke $var sync
```

`lesson7f.jacl` - *Update of application where bindings in EAR file ignored in favor if existing*

Note: The `-update.ignore.new` option must be used in conjunction with the `-update` option. If you forget to include `-update`, the `-update.ignore.new` option will be ignored.

Mapping an application to a data resource

Applications that are designed to map to a data resource such as CICS will have within the EAR file a `<resource-ref>` tag in an XML deployment descriptor. If you were installing that application using the Admin Console, you would be presented with something like this:



WSADMIN function to accomplish the same thing: `MapResRefToEJB`

What Admin Console will present when an application has a `<resource-ref>` in deployment descriptor

Format of `MapResRefToEJB`

This option of the `$AdminApp` install method has six arguments. This was determined by running issuing the following command against the EAR file:

```
$AdminApp taskInfo /u/user1/BeCashAc.ear MapResRefToEJB
```

Note: We are *not* supplying the `BeCashAc.ear` file with this white paper. Consider what follows merely an illustration.

```
WASX7348I: Each element of the MapResRefToEJB task consists of the
following 6 fields:

"module", "EJB", "uri", "referenceBinding", "resRef.type", "JNDI".

Of these fields, the following may be assigned values: "JNDI"
and the following are required: "JNDI"

The current contents of the task after running default bindings are:
module: BeCashAcEJB
EJB: BeCashAcSession
uri: BeCashAcEJB.jar,META-INF/ejb-jar.xml
referenceBinding: CICSConnectionFactory
resRef.type: javax.resource.cci.ConnectionFactory
JNDI: CICSConnectionFactory
```

Six arguments for the `MapResRefToEJB` option

What the deployment descriptor of the EAR file presently has assigned for those values

The six argument fields for the `MapResRefToEJB` option of `install`

Note: You'll find `taskInfo` to be a valuable tool to figure out what's going on inside an EAR file.

The first five fields are used to identify the module and the resource reference. Only the sixth field -- "JNDI" -- may be assigned a value. And the value you assign it is the JNDI name of the connection factory to which you wish to map this module. So, for example, if the JNDI name of the CICS adapter connection factory is `eis/My_CICS_CF`, then the command to install this application would be:

```
$AdminApp install /u/user1/BeCashAc.ear
 {-node g5nodec -server g5sr01c
  -MapResRefToEJB
   {{BeCashAcEJB
     BeCashAcSession
     BeCashAcEJB.jar,META-INF/ejb-jar.xml
     CICSConnectionFactory
     javax.resource.cci.ConnectionFactory
     eis/My_CICS_CF
   }}
 }
```

Command would be on one line in a Jacl script. Broken across lines here to illustrate more clearly the six fields

Example `$AdminApp install` command that maps a resource reference to a CICS conn. factory

Example Jacl script that installs application and maps resource reference

Note: Since we are *not* supplying the `BeCashAc.ear` file, this is given to you simply as an example to use when creating your own Jacl script. Don't run this Jacl script unless you have a resource adapter installed and a connection factory defined.

```
# --------------------------------------------------------------------
# Application and Target settings
# --------------------------------------------------------------------
set ear_file         "/u/user1/BeCashAc.ear"
set server           "<server>"
set node             "<node>"
# --------------------------------------------------------------------
# MapResRefToEJB settings
# --------------------------------------------------------------------
set r_option         "-MapResRefToEJB"
set r_mod            "\"BeCashAcEJB\""
set r_ejb            "\"BeCashAcSession\""
set r_uri            "BeCashAcEJB.jar,META-INF/ejb-jar.xml"
set r_ref            "CICSConnectionFactory "
set r_class          "javax.resource.cci.ConnectionFactory"
set r_jndi           "eis/My_CICS_CF"
# --------------------------------------------------------------------
# Construct MapResRefToEJB option list
# --------------------------------------------------------------------
set r_string "$r_mod $r_ejb $r_uri $r_ref $r_class $r_jndi"
set r_list [list $r_string]
# --------------------------------------------------------------------
# Construct $AdminApp command
# --------------------------------------------------------------------
set cmd_opt_list [list -node $node -server $server]
lappend cmd_opt_list $r_option
lappend cmd_opt_list $r_list
$AdminApp install $ear_file $cmd_opt_list
$AdminConfig save
# --------------------------------------------------------------------
# Invoke node synchronization
# --------------------------------------------------------------------
set var [$AdminControl completeObjectName type=NodeSync,node=$node,*]
$AdminControl invoke $var sync
```

Use taskInfo to determine values from *your* EAR file.

`lesson7g.jacl` - *Mapping resource reference*

Concluding points on this lesson

The `$AdminApp install` command will likely be the one you'll use most productively at first. As you have seen, the options that may be employed with the `install` method can be somewhat complex. But two other methods help in this process:

- The `options` method may be run against an EAR file to determine what options are applicable to the EAR.
- The `taskInfo` method may be run against an EAR to determine the values presently set within the EAR for a particular task, such as `MapResRefToEJB`, for example.

Note: Every one of the exercises in this section could be invoked from batch JCL. We didn't illustrate that here because we were trying to focus on the commands themselves. Always remember: if you have a Jacl script, it can be invoked from a batch JCL.

Lesson 8: WSADMIN and Clusters

To WebSphere, a cluster is in many ways not that unique a thing. For the purposes of installing an application, for example, a cluster is considered a "target" just like a server is a target. Clusters have long names and short names, just like servers do. That said, clusters *do* have unique characteristics: a cluster may have members on different MVS images in a cell; a cluster may be in a state somewhere between "started" and "stopped" -- if one cluster member us up but another is down, the cluster is in a "partial start" status.

In this lesson we'll make use of all the objects -- `$AdminApp`, `$AdminConfig`, `$AdminControl` -- as we do a survey of things WSADMIN can do with clusters.

What clusters are in your environment?

The following command will list out the clusters found in the configuration repository:

```
$AdminConfig list ServerCluster
```

Lists all the clusters found in the configuration repository

```
g5sr02cluster(cells/g5cell/clusters/g5sr02cluster:cluster.xml#g5sr02cluster)
```

The "config ID" of the cluster

Listing the clusters in a configuration

What servers are members of that cluster?

There are quite a few different ways you can display the members of a cluster, once you know the name of the cluster. Perhaps the most useful is this:

```
set cluster_id [$AdminConfig getid /ServerCluster:g5sr02cluster/]
$AdminConfig list ClusterMember $cluster_id

g5sr02c(cells/g5cell/clusters/g5sr02cluster:cluster.xml#ClusterMember_1081342348205)
g5sr02d(cells/g5cell/clusters/g5sr02cluster:cluster.xml#ClusterMember_1081365182924)
```

The long names of the members in this cluster

Listing the members of a cluster, once you've captured the "config ID" of the cluster

What nodes are those cluster members configured?

Here we dig a little deeper and display the attributes of each cluster member:

```
set cluster_id [$AdminConfig getid /ServerCluster:g5sr02cluster/]
set mem_one [lindex $cluster_id 0]
$AdminConfig show $mem_one

{cluster g5sr02cluster(cells/g5cell/clusters/g5sr02cluster:cluster.xml#g5sr02cluster)}
{memberName g5sr02c}
{nodeName g5nodec}
{uniqueId BB07FDFD4E35D4550000013800000000109521847}
{weight 2}
```

The node in which that cluster member resides

Using "show" to display the attributes of a cluster member

Installing an application into a cluster

This is really no different from what we've shown in the past, except the "target" of the installation is no longer `{-node <node> -server <server>}`, but rather simply a pointer to the cluster:

```
$AdminApp install /u/user1/MyIVT.ear {-cluster <cluster>}
$AdminConfig save
```

`lesson8a.jacl` - *Installing an application into a cluster*

Caution: We've not "synched" to the nodes in that Jacl script. The synchronization command we've used so far in this document was good for only one node. If your entire cluster is contained within one node, what we've shown before will work here as well:

```
set var [$AdminControl completeObjectName type=NodeSync,node=<node>,*]
$AdminControl invoke $var sync
```

But if your cluster spans nodes (the more likely case), then you have to synch to each node individually.

Manually synchronizing with each known node of the cluster

This method is the most simple: duplicate the two command lines that synchronize to a node and change the node name in the second set of commands. That way the synchronization process occurs twice:

```
$AdminApp install /u/user1/MyIVT.ear {-cluster g5sr02cluster}
$AdminConfig save
set var [$AdminControl completeObjectName type=NodeSync,node=g5nodec,*]
$AdminControl invoke $var sync
set var [$AdminControl completeObjectName type=NodeSync,node=g5noded,*]
$AdminControl invoke $var sync
```

Synch once to node "g5nodec," then synch again but to the second node: "g5noded"

`lesson8b.jacl` - *Manually synchronizing to the two known nodes in which the cluster is configured*

Note: That's not very elegant, but it is effective. Would it be possible to programmatically extract a list of the nodes and iterate through the list to complete a cluster-wide synchronization? Yes. That's coming up .

Programmatically synchronizing every node

We saw how to do this back in "Programmatically synchronizing with every node in the cell" on page 51.

Programmatically synchronizing to just the cluster nodes

This is a variation on the cell-wide synchronization example shown on page 51. In this Jacl script, the cluster's long name is used to get the ID of the cluster, then each cluster member is queried for the `nodeName` attribute:

```
      1
   set c_id     [$AdminConfig getid /ServerCluster:g5sr02cluster/]
2  set c_membs  [$AdminConfig list ClusterMember $c_id]
   foreach m_id $c_membs {
    set node_name [$AdminConfig showAttribute $m_id nodeName] 4
3   set nodeSync [$AdminControl completeObjectName type=NodeSync,node=$node_name,*]
    set work [$AdminControl invoke $nodeSync sync]  5
   }
```

Synchronizing only those nodes in which cluster members reside

??? That script is provided in the sample file `synch_cluster.jacl`

1 The "config ID" of the named cluster is put into the variable `$c_id`.

2 A list variable (`$c_membs`) is populated with the config IDs of each cluster member in the cluster

3 The foreach loop then processes through the cluster members in the variable `$c_membs`. Each iteration places that iteration's cluster member ID into the variable `$m_id`.

4 The `nodeName` attribute of that cluster member is extracted and placed into variable `$node_name`.

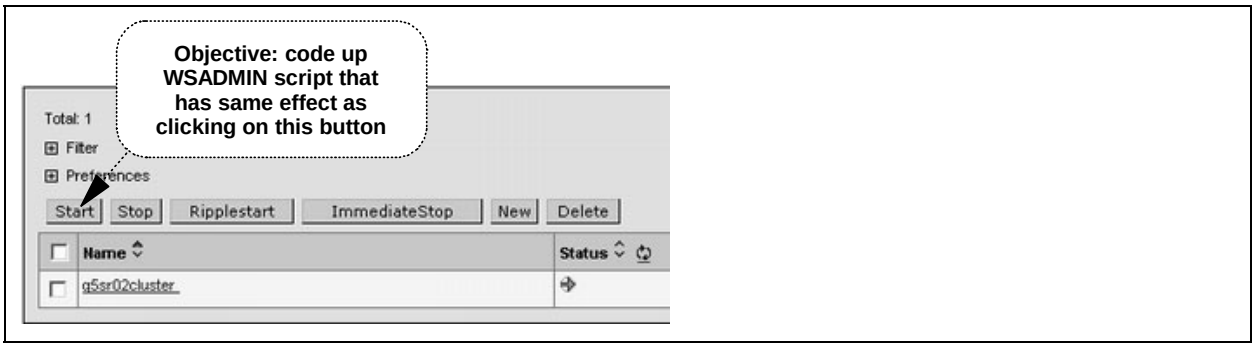
5 The node is synchronized.

Note: This Jacl example has its own shortcomings: if you have a vertical cluster with all cluster members in the same node -- a not very practical use of clusters, but something you may do in a test cell -- this Jacl script will synchronize the same node over and over again. That won't hurt anything other than burning extra cycles.

Starting the members of a cluster

Back in "Starting a server in a Network Deployment configuration" on page 75 we saw how to start an individual server using the `$AdminControl` object. You may start the members of a cluster individually by using the mechanism described on page 75. There's nothing about being a member of a cluster that prevents you from starting or stopping the members individually.

But what we really want to do is issue a "start" against the cluster as a whole, like what's possible from the Admin Console:



Admin Console has button that permits starting cluster as a whole. Objective: get WSADMIN to do same

This is accomplished in a way similar to the synchronizing of nodes. The script to accomplish this would look like this:

```
                                                    The long name of the
                                                    cluster you wish to start

set cluster [$AdminControl completeObjectName type=Cluster,name=g5sr02cluster,*]
$AdminControl invoke $cluster start
```

lesson8c.jacl - *Starting a cluster*

Stopping the members of a cluster

The `stop` operation is invoked:

```
                                                    The long name of the
                                                    cluster you wish to stop

set cluster [$AdminControl completeObjectName type=Cluster,name=g5sr02cluster,*]
$AdminControl invoke $cluster stop
```

lesson8d.jacl - *Starting a cluster*

Checking the status of the cluster and cluster members

To check the status of the *cluster as a whole*, use the `$AdminControl getAttribute` method and check the `state`:

```
                                                    The long name of the
                                                    cluster you wish to query

set cluster [$AdminControl completeObjectName type=Cluster,name=g5sr02cluster,*]
$AdminControl getAttribute $cluster state

            websphere.cluster.partial.start
            websphere.cluster.running
            websphere.cluster.stopping
            websphere.cluster.stopped
                                                    States of the server
```

lesson8e.jacl - *Checking the state of a cluster*

To check the status of individual members of the cluster, you may simply query the state of the individual server, as illustrated back in "Checking the status of a server process" on page 77.

Appendix A: Exercises (available for copy-and-paste)

Lesson 2 Exercises

lesson2a.jacl

```
set list [$AdminApp list]
puts stdout $list
```

LESSON2B.JCL

```
//*********************************************
//*  STEP 1 - Invoke WSADMIN, issue command
//*********************************************
//INST1 EXEC PGM=IKJEFT01,REGION=0M
//SYSTSPRT DD  SYSOUT=*
//SYSTSIN  DD  *
 BPXBATCH SH +
   /wasv5config/azcell+
   /DeploymentManager+
   /bin/wsadmin.sh -conntype none +
   -c '$AdminApp list' +
   1> /tmp/lesson2b.out +
   2> /tmp/lesson2b.err
/*
//************************************
//*  STEP Copy - Copy script output back to joblog
//*********************************************
//DIAPPC EXEC PGM=IKJEFT01,REGION=0M
//SYSEXEC  DD  DISP=SHR,DSN=WAS502.WAS.SBBOEXEC
//SYSTSIN  DD *
   BBOHFSWR '/tmp/lesson2b.out'
   BBOHFSWR '/tmp/lesson2b.err'
//SYSTSPRT DD SYSOUT=*
//
```

lesson2c.jacl

```
set list [$AdminApp list]
puts stdout $list
```

LESSON2C.JCL

```
//*********************************************
//*  STEP 1 - Invoke WSADMIN, point to file
//*********************************************
//INST1 EXEC PGM=IKJEFT01,REGION=0M
//SYSTSPRT DD  SYSOUT=*
//SYSTSIN  DD  *
 BPXBATCH SH +
   /wasv5config/azcell+
   /DeploymentManager+
   /bin/wsadmin.sh +
   -javaoption -Dscript.encoding=Cp1047 +
   -conntype none +
   -f /u/user1/lesson2c.jacl +
   1> /tmp/lesson2c.out +
   2> /tmp/lesson2c.err
/*
//*********************************************
//*  STEP Copy - Copy script output back to joblog
//*********************************************
//DIAPPC EXEC PGM=IKJEFT01,REGION=0M
//SYSEXEC  DD  DISP=SHR,DSN=WAS502.WAS.SBBOEXEC
//SYSTSIN  DD *
   BBOHFSWR '/tmp/lesson2c.out'
   BBOHFSWR '/tmp/lesson2c.err'
//SYSTSPRT DD SYSOUT=*
//
```

Lesson 3 Exercises

lesson3a.jacl

```
set server "Server1"
puts stdout $server
```

lesson3b.jacl

```
set cell    "mycell"
set node    "mynode"
set server  "Server1"
set appl    "my_appl"
puts stdout "C:$cell N:$node S:$server A:$appl"
```

lesson3c.jacl

```
set string  "Four Score and Seven"
set first   [lindex $string 0]
set second  [lindex $string 1]
set third   [lindex $string 2]
set fourth  [lindex $string 3]
puts stdout "1st:$first 2nd:$second 3rd:$third 4th:$fourth"
```

lesson3d.jacl

```
set cell    [lindex $argv 0]
set node    [lindex $argv 1]
set server  [lindex $argv 2]
set appl    [lindex $argv 3]
puts stdout "C:$cell N:$node S:$server A:$appl"
```

lesson3e.jacl

```
if { !($argc==3) } then {
  puts stdout "You supplied $argc parameters, not three.  Try again"
} else  {
 set cell   [lindex $argv 0]
 set node   [lindex $argv 1]
 set server [lindex $argv 2]
 puts stdout "Parameters: cell:$cell node:$node server:$server"
}
```

lesson3f.jacl

```
set node "<node>"
set server "<server>"
set ear "/u/user1/MyIVT.ear"
set opt [list -node $node -server $server]
puts stdout "AdminApp install $ear $opt"
```

lesson3g.jacl

```
set inner [list opt1a opt1b opt1c]
set outer [list -opt1 $inner]
puts stdout "task $outer"
```

lesson3h.jacl

```
set inopt1 [list opt1a opt1b]
set inopt2 [list opt2a opt2b]
set outer [list -opt1 $inopt1 -opt2 $inopt2]
puts stdout "task $outer"
```

Lesson 4 Exercises

lesson4a.jacl

```
$AdminApp install /u/user1/MyIVT.ear {-node <node> -server <server>}
$AdminConfig save
```

lesson4b.jacl

```
$AdminApp uninstall My_IVT_Application
$AdminConfig save
```

lesson4c.jacl

```
$AdminApp install /u/user1/MyIVT.ear {-node <node> -server <server> -MapWebModToVH
{{"My_IVT_Webapp_Display_Name" MyIVTWebApp.war,WEB-INF/web.xml WSC_host}}}
$AdminConfig save
```

lesson4d.jacl

```
set ear     "/u/user1/MyIVT.ear"
set node    "<node>"
set server  "<server>"
set WebMod  "\"My_IVT_Webapp_Display_Name\""
set uri     "MyIVTWebApp.war,WEB-INF/web.xml"
set VH      "WSC_host"
# ------------------------------------------------------------------
set VHlist  "$WebMod $uri $VH"
set VHopts  [list $VHlist]
set options [list -node $node -server $server -MapWebModToVH $VHopts]
# ------------------------------------------------------------------
$AdminApp install $ear $options
$AdminConfig save
```

lesson4e.jacl

```
set parm [lindex $argv 0]
if { !($parm == 1 || $parm == 2) } then {
 puts stdout "Must supply either 1 for install or 2 for uninstall"
 exit
}
# EAR FILE, OPTIONS ------------------------------------------
set ear      "/u/user1/MyIVT.ear"
set name     "My_IVT_Application"
set node     "<node>"
set server   "<server>"
set opts     [list -node $node -server $server]
# INSTALL/UNINSTALL ------------------------------------------
if { ($parm == 1) } then {
 puts stdout "Installing $name"
 $AdminApp install $ear $opts
 $AdminConfig save
} else {
 puts stdout "Uninstalling $name"
 $AdminApp uninstall $name
 $AdminConfig save
}
# ------------------------------------------------------------
puts stdout "All done."
```

Lesson 5 Exercises

synch_all.jacl

```
set node_ids [$AdminConfig list Node]
foreach node $node_ids {
 set node_name [$AdminConfig showAttribute $node name]
 set nodeSync  [$AdminControl completeObjectName type=NodeSync,node=$node_name,*]
 if { !($nodeSync=="") } then {
  $AdminControl invoke $nodeSync sync
 }
}
```

lesson5a.jacl

```
set cellid [$AdminConfig getid /Cell:<cell>/]
puts stdout $cellid
```

lesson5b.jacl

```
set cellid [$AdminConfig getid /Cell:<cell>/]
$AdminConfig create VirtualHost $cellid {{name Test_VH}}
$AdminConfig save
```

lesson5c.jacl

```
set vh_id [$AdminConfig getid /VirtualHost:default_host/]
puts stdout "ID of VirtualHost is $vh_id"
```

lesson5d.jacl

```
set vh_id [$AdminConfig getid /VirtualHost:default_host/]
set show_out [$AdminConfig show $vh_id]
puts stdout "Attributes are $show_out"
```

lesson5e.jacl

```
set vh_id [$AdminConfig getid /VirtualHost:default_host/]
set show_out [$AdminConfig showAttribute $vh_id aliases]
puts stdout "Attributes are $show_out"
```

lesson5f.jacl

```
set alias_id "(cells/<cell>:virtualhosts.xml#HostAlias_1)"
puts stdout [$AdminConfig show $alias_id]
```

lesson5g.jacl

```
set alias_id "(cells/<cell>:virtualhosts.xml#HostAlias_1)"
puts stdout [$AdminConfig showAttribute $alias_id port]
```

lesson5h.jacl

```
set cell_id [$AdminConfig getid /Cell:<cell>/]
$AdminConfig create VirtualHost $cell_id {{name New_VH} {aliases {{{port 7777} {hostname *}}}}}
$AdminConfig save
puts stdout [$AdminConfig list VirtualHost]
```

lesson5i.jacl

```
set cell    "<cell>"
set vh_name "New_VH2"
set host1   "*"
set port1   "8888"
# ----------------------------------------------------
set cell_id [$AdminConfig getid /Cell:$cell/]
# ----------------------------------------------------
set name         [list "name" $vh_name]
set p1           [list port $port1]
set h1           [list hostname $host1]
set pair1        [list $p1 $h1]
set alias_attrs  [list $pair1]
set aliases      [list aliases $alias_attrs]
set VH_attrs     [list $name $aliases]
# ----------------------------------------------------
$AdminConfig create VirtualHost $cell_id $VH_attrs
$AdminConfig save
# ----------------------------------------------------
puts stdout [$AdminConfig list VirtualHost]
```

lesson5i-2.jacl

```
set cell    "<cell>"
set vh_name "New_VH3"
set host1   "*"
set port1   "8888"
set host2   "www.myhost.com"
set port2   "9999"
# ----------------------------------------------------
set cell_id [$AdminConfig getid /Cell:$cell/]
# ----------------------------------------------------
```

```
set name         [list "name" $vh_name]
set p1           [list port $port1]
set h1           [list hostname $host1]
set pair1        [list $p1 $h1]
set p2           [list port $port2]
set h2           [list hostname $host2]
set pair2        [list $p2 $h2]
set alias_attrs  [list $pair1 $pair2]
set aliases      [list aliases $alias_attrs]
set VH_attrs     [list $name $aliases]
# ------------------------------------------------------
$AdminConfig create VirtualHost $cell_id $VH_attrs
$AdminConfig save
# ------------------------------------------------------
puts stdout [$AdminConfig list VirtualHost]
```

lesson5j.jacl

```
set exist_name "New_VH2"
set new_name   "Mod_VH2"
# ------------------------------------------------------------
set vh_id   [$AdminConfig getid /VirtualHost:$exist_name/]
# ------------------------------------------------------------
set name_list  [list name $new_name]
set attr_list  [list $name_list]
# ------------------------------------------------------------
$AdminConfig modify $vh_id $attr_list
$AdminConfig save
# ------------------------------------------------------------
puts stdout [$AdminConfig list VirtualHost]
```

lesson5k.jacl

```
set vh_name    "New_VH2"
set port1      "5555"
set host1      "*"
set port2      "6666"
set host2      "*"
# ------------------------------------------------------------
set vh_id   [$AdminConfig getid /VirtualHost:$vh_name/]
# ------------------------------------------------------------
set p1         [list port $port1]
set h1         [list hostname $host1]
set pair1      [list $p1 $h1]
set p2         [list port $port2]
set h2         [list hostname $host2]
set pair2      [list $p2 $h2]
set pair_list  [list $pair1 $pair2]
set alias_list [list aliases $pair_list]
set attr_list  [list $alias_list]
# ------------------------------------------------------------
$AdminConfig modify $vh_id $attr_list
$AdminConfig save
# ------------------------------------------------------------
puts stdout [$AdminConfig list VirtualHost]
```

lesson5l.jacl

```
set vh_name    "New_VH2"
set vh_id      [$AdminConfig getid /VirtualHost:$vh_name/]
# ------------------------------------------------------------
$AdminConfig remove $vh_id
$AdminConfig save
# ------------------------------------------------------------
puts stdout [$AdminConfig list VirtualHost]
```

lesson5m.jacl

```
set model_serv   [$AdminConfig getid /Server:<serve_long>/]
set parent_node  [$AdminConfig getid /Node:<node_long>/]
$AdminConfig createUsingTemplate Server $parent_node {{name <serve_long>}} $model_serv
$AdminConfig save
```

lesson5n.jacl

```
set model_serv      "<model_server_long>"
set new_serv_name   "<new_server_long>"
set parent          "<node_long>"
# ------------------------------------------------------------
set m_ID            [$AdminConfig getid /Server:$model_serv/]
set p_ID            [$AdminConfig getid /Node:$parent/]
# ------------------------------------------------------------
set name_pair       [list name $new_serv_name]
set attr_list       [list $name_pair]
# ------------------------------------------------------------
$AdminConfig createUsingTemplate Server $p_ID $attr_list $m_ID
$AdminConfig save
# ------------------------------------------------------------
set var [$AdminControl completeObjectName type=NodeSync,node=$parent,*]
$AdminControl invoke $var sync
```

lesson5o-pre.jacl

```
set server_long  "<new_server_long>"
set server_id    [$AdminConfig getid /Server:$server_long/]
# ------------------------------------------------------
puts stdout [$AdminConfig show $server_id]
```

lesson5o.jacl

```
set server_long  "<server_long>"
set new_short    "<server_short>"
set parent       "<node_long>"
set server_id    [$AdminConfig getid /Server:$server_long/]
# ------------------------------------------------------
set name_list    [list shortName $new_short]
set attr_list    [list $name_list]
# ------------------------------------------------------
$AdminConfig modify $server_id $attr_list
$AdminConfig save
# ------------------------------------------------------
set var [$AdminControl completeObjectName type=NodeSync,node=$parent,*]
$AdminControl invoke $var sync
```

lesson5p.jacl

```
set server_long   "<server_long>"
set parent        "<node_long>"
set new_CTN       "<NEW_CLUSTER_TRAN>"
# ------------------------------------------------------------
set server_ID     [$AdminConfig getid /Server:$server_long/]
set appl_serv_ID  [$AdminConfig list ApplicationServer $server_ID]
set prop_list     [$AdminConfig list Property $appl_serv_ID]
set ctn_prop      [lindex $prop_list 0]
set first_seven   [string range $ctn_prop 0 6]
# ------------------------------------------------------------
if { !($first_seven=="Cluster") } then {
 puts stdout "Got wrong property!"
 exit
}
# ------------------------------------------------------------
set value_pair    [list value $new_CTN]
set attr_list     [list $value_pair]
$AdminConfig modify $ctn_prop $attr_list
$AdminConfig save
# ------------------------------------------------------------
set var [$AdminControl completeObjectName type=NodeSync,node=$parent,*]
$AdminControl invoke $var sync
```

Lesson 6 Exercises

LESSON6A.JCL

```
//************************************************
//*  STEP 1 - Start Server
//************************************************
//INST1 EXEC PGM=IKJEFT01,REGION=0M
//SYSTSPRT DD  SYSOUT=*
//SYSTSIN  DD  *
 BPXBATCH SH +
   /<config root>+
   /DeploymentManager+
   /bin/wsadmin.sh +
   -conntype SOAP +
   -host <host> +
   -port <port> +
   -c '$AdminControl startServer <server> <node>' +
   1> /tmp/lesson6a.out +
   2> /tmp/lesson6a.err
/*
//************************************************
//*  STEP Copy - Copy script output back to joblog
//************************************************
//DIAPPC EXEC PGM=IKJEFT01,REGION=0M
//SYSEXEC  DD  DISP=SHR,DSN=WAS502.WAS.SBBOEXEC
//SYSTSIN  DD *
   BBOHFSWR '/tmp/lesson6a.out'
   BBOHFSWR '/tmp/lesson6a.err'
//SYSTSPRT DD SYSOUT=*
//
```

lesson6b.jacl

```
set server [$AdminControl completeObjectName
                                        cell=<cell>,node=<node>,name=<server>,type=Server,*]
puts stdout [$AdminControl getAttribute $server state]
```

lesson6c.jacl

```
set appManager [$AdminControl queryNames
                          cell=<cell>,node=<node>,type=ApplicationManager,process=<server>,*]
$AdminControl invoke $appManager startApplication My_IVT_Application
```

lesson6d.jacl

```
set appManager [$AdminControl queryNames
                          cell=<cell>,node=<node>,type=ApplicationManager,process=<server>,*]
$AdminControl invoke $appManager stopApplication My_IVT_Application
```

lesson6e.jacl

```
set app_name  "My_IVT_Application"
# ---------------------------------------------
set status    [$AdminControl completeObjectName type=Application,name=$app_name,*]
# ---------------------------------------------
set length    [string length $status]
if { ($length==0) } then {
 puts stdout "Application is STOPPED"
} else {
 puts stdout "Application is STARTED"
}
```

Lesson 7 Exercises

lesson7a.jacl

```
$AdminApp install /u/user1/MyIVT.ear {-node <node> -server <server>
                                                          -appname My_IVT_Application2}
$AdminConfig save
set var [$AdminControl completeObjectName type=NodeSync,node=<node>,*]
$AdminControl invoke $var sync
```

lesson7b.jacl

```
$AdminApp install /u/user1/MyIVT.ear {-node <node> -server <server>
    -BindJndiForEJBNonMessageBinding {{"My IVT EJB Module Display Name" My_IVT_EJB_Name
     MyIVTStatelessSession.jar,META-INF/ejb-jar.xml ejb/New_JNDI}}}
set var [$AdminControl completeObjectName type=NodeSync,node=<node>,*]
$AdminControl invoke $var sync
```

lesson7c.jacl

```
$AdminApp install /u/user1/MyIVT.ear {-node <node> -server <server>
    -MapEJBRefToEJB {{"My_IVT_Webapp_Display_Name" "" MyIVTWebApp.war,WEB-INF/web.xml
     ejb/ivtEJBObject com.ibm.websphere.ivt.ivtEJB.ivtEJBObject ejb/New_JNDI}}}
set var [$AdminControl completeObjectName type=NodeSync,node=<node>,*]
$AdminControl invoke $var sync
```

lesson7d.jacl

```
# ------------------------------------------------------------------
# Application and Target settings
# ------------------------------------------------------------------
set ear_file         "/u/user1/MyIVT.ear"
set server           "<server>"
set node             "<node>"
# ------------------------------------------------------------------
# BindJndiForEJBNonMessageBinding settings
# ------------------------------------------------------------------
set b_option         "-BindJndiForEJBNonMessageBinding"
set b_mod            "\"My IVT EJB Module Display Name\""
set b_ejb            "My_IVT_EJB_Name"
set b_uri            "MyIVTStatelessSession.jar,META-INF/ejb-jar.xml"
set b_jndi           "ejb/New_JNDI"
# ------------------------------------------------------------------
# MapEJBRefToEJB settings
# ------------------------------------------------------------------
set m_option         "-MapEJBRefToEJB"
set m_mod            "\"My_IVT_Webapp_Display_Name\""
set m_ejb            "\"\""
set m_uri            "MyIVTWebApp.war,WEB-INF/web.xml"
set m_ref            "ejb/ivtEJBObject"
set m_class          "com.ibm.websphere.ivt.ivtEJB.ivtEJBObject"
set m_jndi           $b_jndi
# ------------------------------------------------------------------
# Construct BindJndiForEJBNonMessageBinding option list
# ------------------------------------------------------------------
set b_string "$b_mod $b_ejb $b_uri $b_jndi"
set b_list [list $b_string]
# ------------------------------------------------------------------
# Construct MapEJBRefToEJB option list
# ------------------------------------------------------------------
set m_string "$m_mod $m_ejb $m_uri $m_ref $m_class $m_jndi"
set m_list [list $m_string]
# ------------------------------------------------------------------
# Construct $AdminApp command
# ------------------------------------------------------------------
set cmd_opt_list [list -node $node -server $server]
lappend cmd_opt_list $b_option
lappend cmd_opt_list $b_list
lappend cmd_opt_list $m_option
lappend cmd_opt_list $m_list
$AdminApp install $ear_file $cmd_opt_list
$AdminConfig save
# ------------------------------------------------------------------
# Invoke node synchronization
# ------------------------------------------------------------------
set var [$AdminControl completeObjectName type=NodeSync,node=$node,*]
$AdminControl invoke $var sync
```

lesson7e.jacl

```
$AdminApp install /u/user1/MyIVT.ear {-node <node> -server <server>
                                                    -update -appname My_IVT_Application}
$AdminConfig save
set var [$AdminControl completeObjectName type=NodeSync,node=<node>,*]
$AdminControl invoke $var sync
```

lesson7f.jacl

```
$AdminApp install /u/user1/MyIVT.ear {-node <node> -server <server>
                                    -update -update.ignore.new -appname My_IVT_Application}
$AdminConfig save
set var [$AdminControl completeObjectName type=NodeSync,node=<node>,*]
$AdminControl invoke $var sync
```

lesson7g.jacl

```
# ------------------------------------------------------------------
# Application and Target settings
# ------------------------------------------------------------------
set ear_file         "/u/user1/BeCashAc.ear"
set server           "<server>"
set node             "<node>"
# ------------------------------------------------------------------
# MapResRefToEJB settings
# ------------------------------------------------------------------
set r_option         "-MapResRefToEJB"
set r_mod            "\"BeCashAcEJB\""
set r_ejb            "\"BeCashAcSession\""
set r_uri            "BeCashAcEJB.jar,META-INF/ejb-jar.xml"
set r_ref            "CICSConnectionFactory "
set r_class          "javax.resource.cci.ConnectionFactory"
set r_jndi           "eis/My_CICS_CF"
# ------------------------------------------------------------------
# Construct MapResRefToEJB option list
# ------------------------------------------------------------------
set r_string "$r_mod $r_ejb $r_uri $r_ref $r_class $r_jndi"
set r_list [list $r_string]
# ------------------------------------------------------------------
# Construct $AdminApp command
# ------------------------------------------------------------------
set cmd_opt_list [list -node $node -server $server]
lappend cmd_opt_list $r_option
lappend cmd_opt_list $r_list
$AdminApp install $ear_file $cmd_opt_list
$AdminConfig save
# ------------------------------------------------------------------
# Invoke node synchronization
# ------------------------------------------------------------------
set var [$AdminControl completeObjectName type=NodeSync,node=$node,*]
$AdminControl invoke $var sync
```

Lesson 8 Exercises

lesson8a.jacl

```
$AdminApp install /u/user1/MyIVT.ear {-cluster <cluster>}
$AdminConfig save
```

lesson8b.jacl

```
$AdminApp install /u/user1/MyIVT.ear {-cluster <cluster>}
$AdminConfig save
set var [$AdminControl completeObjectName type=NodeSync,node=<node_1>,*]
$AdminControl invoke $var sync
set var [$AdminControl completeObjectName type=NodeSync,node=<node_2>,*]
$AdminControl invoke $var sync
```

synch_cluster.jacl

```
set c_id     [$AdminConfig getid /ServerCluster:<cluster_long>/]
set c_membs  [$AdminConfig list ClusterMember $c_id]
foreach m_id $c_membs {
 set node_name [$AdminConfig showAttribute $m_id nodeName]
 set nodeSync [$AdminControl completeObjectName type=NodeSync,node=$node_name,*]
 set work [$AdminControl invoke $nodeSync sync]
}
```

lesson8c.jacl

```
set cluster [$AdminControl completeObjectName type=Cluster,name=<cluster_long>,*]
$AdminControl invoke $cluster start
```

lesson8d.jacl

```
set cluster [$AdminControl completeObjectName type=Cluster,name=<cluster_long>,*]
$AdminControl invoke $cluster stop
```

lesson8e.jacl

```
set cluster [$AdminControl completeObjectName type=Cluster,name=<cluster_long>,*]
$AdminControl getAttribute $cluster state
```

More Information

(under construction)

Document Change History

May 5, 2004	Original preliminary draft document

Index

End of Document WP100421