WebSphere Message Broker

CMP Programming

Version 6 Release 0

WebSphere Message Broker

CMP Programming

Version 6 Release 0

Note
Before using this information and the product it supports, read the information in the Notices appendix.

First Edition (September 2005)

This edition applies to IBM® WebSphere® Message Broker Version 6.0 and to all subsequent releases and modifications until otherwise indicated in new editions.

Contents

About this topic collection

This PDF has been created from the WebSphere Message Broker Version 6.0 GA (September 2005) information center topics. Always refer to the WebSphere Message Broker online information center to access the most current information. The information center is periodically updated on the document update site and this PDF and others that you can download from that Web site might not contain the most current information.

The topic content included in the PDF does not include the ″Related Links″ sections provided in the online topics. Links within the topic content itself are included, but are active only if they link to another topic in the same PDF collection. Links to topics outside this topic collection are also shown, but these attempt to link to a PDF that is called after the topic identifier (for example, ac12340_.pdf) and therefore fail. Use the online information to navigate freely between topics.

Feedback: do not provide feedback on this PDF. Refer to the online information to ensure that you have access to the most current information, and use the Feedback link that appears at the end of each topic to report any errors or suggestions for improvement. Using the Feedback link provides precise information about the location of your comment.

The content of these topics is created for viewing online; you might find that the formatting and presentation of some figures, tables, examples, and so on are not optimized for the printed page. Text highlighting might also have a different appearance.

Part 1. Developing applications using the CMP

Developing applications that use the Configuration Manager Proxy Java API

There are many tasks involved in developing Configuration Manager Proxy (CMP):
- "Configuring an environment for developing and running Configuration Manager Proxy applications" on page 11
- "Connecting to the Configuration Manager using the Configuration Manager Proxy" on page 14
- "Navigating broker domains using the Configuration Manager Proxy" on page 16
- "Managing broker domains using the Configuration Manager Proxy" on page 25

A number of samples are provided to demonstrate simple CMP scenarios. Run and explore the samples to learn about what you can do with the CMP. See "The Configuration Manager Proxy samples" on page 5.

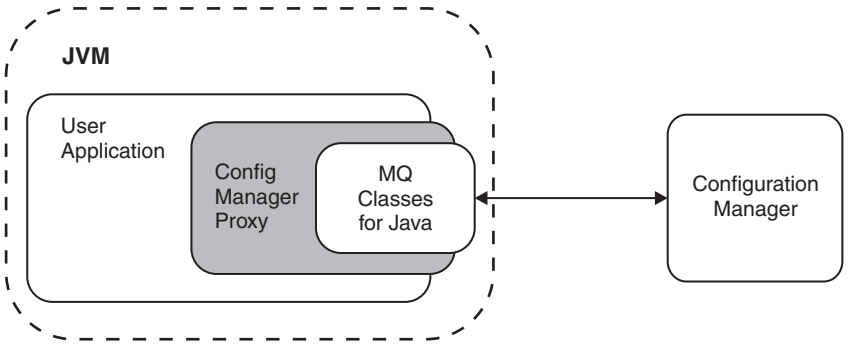
When you have finished:
- You can debug your message flow using the flow debugger. For more details, see Testing and debugging message flow applications.
- You can deploy your message flow to one or more production brokers. See Deploying for further information.

Configuration Manager Proxy

The Configuration Manager Proxy (CMP) is an application programming interface that your applications can use to control broker domains through a remote interface to the Configuration Manager.

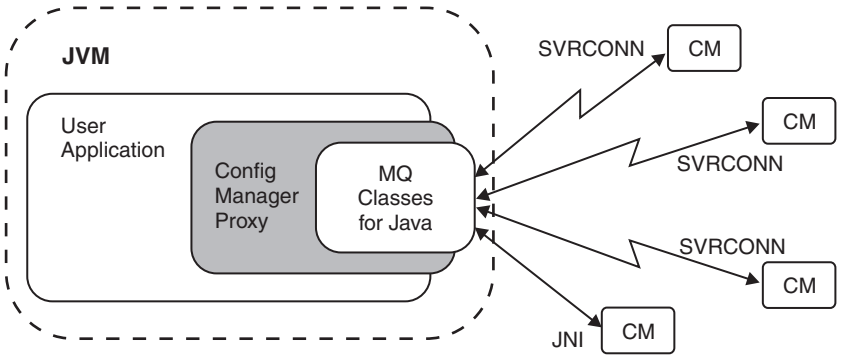
Your applications have complete access to the Configuration Manager functions and resources through the set of Java classes that constitute the CMP. For example, you can use the CMP to interact with the Configuration Manager to:
- Deploy BAR files, Publish/Subscribe topology, topic trees and broker configuration.
- Modify the Publish/Subscribe topology; add and remove brokers, broker connections and collectives.
- Create, modify, and delete execution groups
- Enquire and set status of objects in the domain, for example, run state, and be informed if status changes.
- Manipulate the topics hierarchy.
- View the broker event log and active subscriptions table.
- Modify domain Access Control Lists, when connected to Version 6.0 Configuration Managers only.

The CMP is a lightweight set of Java classes that sit logically between the user application and the Configuration Manager, inside the Java Virtual Machine (JVM) of the user application. It requires the WebSphere MQ Classes for Java in order to function, as shown below.



The CMP application can be on the same physical machine as the Configuration Manager (JNI to the queue manager using the WebSphere MQ Java Bindings transport) or distributed over a TCP/IP network (a WebSphere MQ SVRCONN channel using the WebSphere MQ Java Client transport).

It is possible for the CMP to communicate with multiple Configuration Managers from within the same application:



Using the API it is possible to connect to and manipulate Configuration Managers of the following products:

- IBM WebSphere Business Integration Event Broker Version 5.0
- IBM WebSphere Business Integration Message Broker Version 5.0
- IBM WebSphere Business Integration Message Broker Version 5.0 with Rules and Formatter Extension
- WebSphere Event Broker Version 6.0
- WebSphere Message Broker Version 6.0

A domain controlled by a Version 5.0 Configuration Manager can consist of Version 2.1 and Version 5.0 brokers, to which either version can be deployed by the CMP. Note also, that although it is only possible to run one Version 5.0 Configuration Manager on each physical machine, a single CMP application can still connect to multiple Version 5.0 Configuration Managers.

The Configuration Manager Proxy samples

Through using the Configuration Manager Proxy (CMP) samples, you can deploy a BAR file, manage a broker domain, or use the CMP API Exerciser to perform various tasks. The CMP samples introduce you, at a basic level, to the features available with the CMP.

Deploy BAR

The Deploy BAR sample attempts to deploy a BAR file to an execution group, and displays the outcome, see "Running the Deploy BAR sample."

Managing a broker domain

The broker domain management sample uses the CMP to display to the screen the complete run state of the domain, see "Running the broker domain management sample" on page 6.

Using the Configuration Manager Proxy API Exerciser

The CMP API Exerciser sample uses the API Exerciser to view and manage a Configuration Manager, customize the API Exerciser, or record and play back configuration scripts, see "Running the Configuration Manager Proxy API Exerciser sample" on page 7.

Modifying a CMP sample

You can modify the CMP samples and change various parameters, which will effect how the sample runs, see "Modifying the Configuration Manager Proxy samples" on page 11.

Running the Deploy BAR sample

Before you start:

The Deploy BAR sample is one of the Configuration Manager Proxy (CMP) samples. The CMP samples can be run as they are shipped, or they can be modified.

- If the Deploy BAR sample has no modifications, there are no prerequisite tasks.
- If the Deploy BAR sample has been modified, the environment much be setup before the sample is run, see "Modifying the Configuration Manager Proxy samples" on page 11.

The source file for this sample is located within the installation directory at /sample/ConfigManagerProxy/cmp/DeployBar.java.

The Deploy BAR sample attempts to deploy a BAR file to an execution group, and displays the outcome. The BAR file, execution group name, and other connection details are hard coded into the application.

Run the Deploy BAR sample.

- On Windows, use the Command Console to execute the following command from the installation directory:

  <INST_DIR>\sample\ConfigManagerProxy\StartDeployBAR.bat

- On other platforms, execute the following shell script from the installation directory:
  <INST_DIR>\sample\ConfigManagerProxy\StartDeployBAR

Where <INST_DIR> is the installation directory.

The default connection parameters used by the sample follow:

Connection parameter	Description
`"localhost"`	Host name of the Configuration Manager.
`1414`	Port of the Configuration Manager.
`"BROKER"`	Queue manager name of the Configuration Manager.
`"BROKER"`	Name of the broker.
`"default"`	Name of the execution group.
`"c://mybar.bar"`	BAR file name to deploy.

The CMP connects to the Configuration Manager running on machine, localhost, on port, 1414, with queue manager, BROKER. Next, the CMP attempts to deploy the file, mybar.bar, to the predefined execution group, default, on broker, BROKER.

Note: The constants that represent the default connection parameters for this sample can be modified, see "Modifying the Configuration Manager Proxy samples" on page 11.

Running the broker domain management sample

Before you start:

The broker domain management sample is one of the Configuration Manager Proxy (CMP) samples. The CMP samples can be run as they are, or they can be modified.

- If the broker domain management sample has no modifications, there are no prerequisite tasks.
- If the broker domain management sample has been modified, the environment much be setup before the sample is run, see "Modifying the Configuration Manager Proxy samples" on page 11.

The source file for this sample is located within the installation directory at /sample/ConfigManagerProxy/cmp/DomainInfo.java.

The broker domain management sample uses the CMP to display to the screen the complete run state of the domain.

Run the broker domain management sample.

- On Windows, use the Command Console to execute the following command from the installation directory, specifying an additional parameter that indicates the Configuration Manager whose domain is to be iterated:
  <INST_DIR>\sample\ConfigManagerProxy\StartDomainInfo.bat <CONFIG_MANAGER>Where <INST_DIR> is the installation directory, and <CONFIG_MANAGER> is the full file path to a Configuration Manager file (with extension .configmgr).

- On other platforms, execute the following shell script from the installation directory, specifying an additional parameter that indicates the Configuration Manager whose domain is to be iterated:

  <INST_DIR>\sample\ConfigManagerProxy\StartDomainInfo <CONFIG_MANAGER>Where <INST_DIR> is the installation directory, and <CONFIG_MANAGER> is the full filepath to a Configuration Manager file (with extension .configmgr).
- An alternative is to run the sample in interactive mode. This causes the sample to listen for changes to the domain.

  To run the sample in interactive mode, specify the -i option. For example, \sample\ConfigManagerProxy\StartDomainInfo.bat c:\myConfigMgr.configmgr -i.

  To stop the sample when running interactively, forcibly terminate it using CTRL+C.

The complete run state of the domain is displayed. For example, the following could be displayed:

```
(13/08/04 15:47:37) Connecting. Please wait...
(13/08/04 15:47:38) Successfully connected to the Configuration Manager's
                    Queue Manager.
(13/08/04 15:47:39) Successfully connected to the Configuration Manager.
(13/08/04 15:47:41) Broker 'BROKER' is running.
(13/08/04 15:47:42)   Execution group 'default' on 'BROKER' is running.
(13/08/04 15:47:43)     Message flow 'flow1' on 'default' on 'BROKER' is
                        running.
(13/08/04 15:47:44) Disconnected.
```

In addition, if running in interactive mode, you could see output such as the following:

```
(13/08/04 15:53:46) Listening for changes to the domain...
```

Running the Configuration Manager Proxy API Exerciser sample

Before you start:

The Configuration Manager Proxy API Exerciser sample is one of the Configuration Manager Proxy (CMP) samples. The CMP samples can be run as they are shipped, or they can be modified.

- If the CMP API Exerciser sample has no modifications, there are no prerequisite tasks.
- If the CMP API Exerciser sample has been modified, the environment much be setup before the sample is run, see "Modifying the Configuration Manager Proxy samples" on page 11.

The source files for this sample are located in the folder <INST_DIR>/sample/ConfigManagerProxy/cmp/exerciser, where <INST_DIR> is the installation directory.

In this sample you can use the API Exerciser to view and manage a Configuration Manager, customize the API Exerciser, or record and play back configuration scripts. See the following:

Viewing and managing a broker domain using the Configuration Manager Proxy API Exerciser

The Configuration Manager Proxy API Exerciser sample can be used to view and manipulate a broker domain using the CMP. To view and manage a broker domain using the CMP, do the following:

1. Start the Configuration Manager Proxy API Exerciser.
   - On Windows, click **Start → IBM WebSphere Message Brokers 6.0 → Java Programming APIs → Configuration Manager Proxy API Exerciser**.
   - On other platforms, run the following shell script from the installation directory:

     <INST_DIR>\sample\ConfigManagerProxy\StartConfigManagerProxyExerciser

     Where <INST_DIR> is the installation directory.

   The Configuration Manager Proxy API Exerciser window is displayed.
2. Connect to a running Configuration Manager by clicking **File → Connect to Configuration Manager...**

   The Connect to a Configuration Manager... dialog is displayed.
3. Enter the connection parameters to the Configuration Manager, then click **Submit**.

   broker domain information is retrieved and displayed in the Configuration Manager Proxy API Exerciser window. You have now connected to the Configuration Manager using the Configuration Manager Proxy API Exerciser.

   The top left of the screen contains a hierarchical representation of the broker domain to which you are connected. Selecting objects in the tree causes the table on its right to change, reflecting the attributes of the selected object. The Method column names CMP methods that can be invoked in your own Java applications, and the Result column indicates the data that would be returned by calling the CMP method on the selected object.
4. Execute a CMP method against a broker object. CMP methods are used to manage objects in a broker domain.
   a. In the navigation tree view, right-click a broker.

      A context-sensitive menu is displayed that shows all the available CMP methods.
   b. Select **List connections**.

      Information is displayed in the log view of the Configuration Manager Proxy API Exerciser window. For example, this information could be as follows:

      ```
      (12/08/04 18:24:45) ----> cmp.exerciser.ClassTesterForBrokerProxy.
                                testListConnections(<B1>)
      (12/08/04 18:24:45) There are no connections defined.
      (12/08/04 18:24:45) <---- cmp.exerciser.ClassTesterForBrokerProxy.
                                testListConnections
      ```

      The first line indicates that the method `cmp.exerciser.ClassTesterForBrokerProxy.testListConnections()` was invoked with the parameter of the AdministeredObject representing the

broker, B1. The second line is some output from the method, and the third line indicates that the method completed.

The available CMP methods are used to manage the broker domain.

During these steps you connected to a broker domain, viewed the domain information, and performed a management task using the Configuration Manager Proxy API Exerciser.

Customizing the Configuration Manager Proxy API Exerciser

To customize the Configuration Manager Proxy API Exerciser, do the following:

1. Start the Configuration Manager Proxy API Exerciser.
   - On Windows, click **Start → IBM WebSphere Message Brokers 6.0 → Java Programming APIs → Configuration Manager Proxy API Exerciser**.
   - On other platforms, run the following shell script from the installation directory:

     <INST_DIR>\sample\ConfigManagerProxy\StartConfigManagerProxyExerciser

     Where <INST_DIR> is the installation directory.

   The Configuration Manager Proxy API Exerciser window is displayed.
2. Customize the Configuration Manager Proxy API Exerciser by selecting any of the following options from the **File** menu.
   a. Optional: Click **File → Discover Subcomponent Tree Recursively**

      Clicking **Discover Subcomponent Tree Recursively** enables or disables this option.
      - If enabled, when the Configuration Manager Proxy API Exerciser connects to a Configuration Manager it will discover as many domain objects as possible.
      - If disabled, only the top level objects are discovered, and you will need to select the context-sensitive option, **Discover subcomponents**, in order to iterate down the tree.

   b. Optional: Click **File → Use Incremental Deployment**

      Clicking **Use Incremental Deployment** enables or disables this option.

      If enabled, all deploy operations will cause a delta or incremental deploy where relevant.

   c. Optional: Click **File → Show Advanced Properties**

      Clicking **Show Advanced Properties** enables or disables this option.
      - If enabled, output from all available methods is displayed in the right hand pane of the Configuration Manager Proxy API Exerciser.
      - If disabled, output from a subset of the available methods is displayed in the right hand pane of the Configuration Manager Proxy API Exerciser.

   d. Optional: Click **File → Connect Using .configmgr Properties File**

      Clicking **Connect Using .configmgr Properties File** enables or disables this option.

      If enabled, when you connect to a Configuration Manager a file dialog is displayed instead of a prompt for the queue manager parameters, security exit parameters, hostname, and port. The file dialog allows you to navigate to a file with a `configmgr` extension which provides the connection parameters to the Configuration Manager.

   e. Optional: Click **File → Enable MQ Java Client Service Trace**

Clicking **Enable MQ Java Client Service Trace** enables or disables this option.

- If enabled, a level 5 service trace of the MQ Classes for Java runs. Initially, a dialog is displayed which allows you to provide a file name to which trace is to be sent.
- If disabled, level 5 service tracing of the MQ Classes for Java is disabled.

f. Optional: Click **File → Enable Config Manager Proxy Service Trace**

Clicking **Enable Config Manager Proxy Service Trace** enables or disables this option.

- If enabled, a service trace of the CMP run. Initially, a dialog is displayed which allows you to provide a file name to which trace is to be sent.
- If disabled, service tracing of the CMP is disabled.

g. Optional: Click **File → Set Timeout Characteristics...**

Specify the time, in seconds, that the Configuration Manager Proxy API Exerciser will wait for responses from the Configuration Manager and brokers. The default is 6 seconds.

Recording and playing back configuration scripts using the Configuration Manager Proxy API Exerciser

The Configuration Manager Proxy API Exerciser sample can be used to record and play back configuration scripts. To record and play back configuration scripts using the Configuration Manager Proxy API Exerciser, do the following:

1. Start the Configuration Manager Proxy API Exerciser.
   - On Windows, click **Start → IBM WebSphere Message Brokers 6.0 → Java Programming APIs → Configuration Manager Proxy API Exerciser**.
   - On other platforms, run the following shell script from the installation directory:

     <INST_DIR>\sample\ConfigManagerProxy\StartConfigManagerProxyExerciser

     Where <INST_DIR> is the installation directory.

   The Configuration Manager Proxy API Exerciser window is displayed.
2. Start recording a script by clicking **Scripting → Record New Script...**.

   The Save dialog is displayed.
3. Type a name for the script file and select an appropriate file location, then click **Save**.
4. Perform a number of actions on a Configuration Manager using the Configuration Manager Proxy API Exerciser.

   In this case the first action performed will be connecting to a Configuration Manager, however you can start recording a script at any point during the management of a Configuration Manager.

   Note: If the script is to be invoked from the command line, a shell window, or a batch file, ensure the fist action performed in the script is connecting to a Configuration Manager.
5. **Optional:** Insert a pause by clicking **Scripting → Insert a pause...**

   A pause causes the Configuration Manager Proxy API Exerciser to wait for a period of time so that responses can be returned before the next action is issued. This is important in order to remove naming conflicts if you are deleting and recreating objects of the same name.

The Insert a pause... dialog is displayed, which allows you to specify the duration of the pause.

6. Stop recording the script by clicking **Scripting → Stop Recording**.
   Information relating to the actions preformed are saved to the script file.
7. To replay the script file, click **Scripting → Play Back Recorded Script...**.
   The Open dialog is displayed.
8. Select the appropriate script file, then click **Open**.
   The script file is replayed.

   Note: You can execute a script file from the command line, a shell window, or from a batch file.

Modifying the Configuration Manager Proxy samples

You can modify the Configuration Manager Proxy (CMP) samples to change various parameters. Once recompiled this will effect how the sample runs.

The constants used in the samples can represent various attributes of the sample, for example the default connection parameters. By modifying them, you can change how the sample runs.

To modify a sample, do the following:

1. Set up the environment, as described in "Configuring an environment for developing and running Configuration Manager Proxy applications."
2. Locate the sample source file.
   The source files for the CMP samples are located in the following directories:

   Deploy BAR sample
   <INST_DIR>/sample/ConfigManagerProxy/cmp/DeployBAR.java

   broker domain management sample
   <INST_DIR>/sample/ConfigManagerProxy/cmp/DomainInfo.java

   Configuration Manager Proxy API Exerciser sample
   <INST_DIR>/sample/ConfigManagerProxy/cmp/exerciser

   Where <INST_DIR> is the installation directory.
3. Open the source file, and modify the appropriate parameters.
4. Recompile the source file.

The sample is now modified, and can be run.

Configuring an environment for developing and running Configuration Manager Proxy applications

Before you start:

The Configuration Manager Proxy (CMP) is a set of Java classes that are packaged in a single JAR file. To run or develop Java applications that use the CMP (CMP applications) the following prerequisites are required in your environment:

- The WebSphere MQ Classes for Java.
  This provides the internal wire protocol for communicating with the Configuration Manager.

- A Version 1.4.2-compatible JDK.

To setup a computer in preparation for building and running CMP applications, you must configure your CLASSPATH so that it includes the WebSphere MQ Classes for Java and the CMP files.

Follow the link for the appropriate environment:

- "Windows command-line environment"
- "Linux, UNIX systems, and z/OS command-line environment"
- "Eclipse environment" on page 13

You can run CMP applications and, therefore, control Configuration Managers on machines that do not have a WebSphere Message Broker product installed. For more information, see "Environments without the broker component installed" on page 13.

Windows command-line environment

To configure the CLASSPATH:

1. Add the WebSphere MQ Classes for Java JARs to your CLASSPATH. (Refer to the *WebSphere MQ Classes for Java* documentation for information on how to do this.)
2. Add the Configuration Manager Proxy (CMP) JAR to your CLASSPATH:

   ```
   set CLASSPATH = %CLASSPATH%;
       %INSTALLPATH%/classes/ConfigManagerProxy.jar
   ```

   where:

   %INSTALLPATH%
   Refers to the installation directory of the product.
3. Similarly, ensure that your Java development directory is also on the CLASSPATH.

Linux, UNIX systems, and z/OS command-line environment

To configure the CLASSPATH:

1. Add the WebSphere MQ Classes for Java JARs to your CLASSPATH. (Refer to the *WebSphere MQ Classes for Java* documentation for information on how to do this.)
2. Add the Configuration Manager Proxy (CMP) JAR to your CLASSPATH:

   ```
   export CLASSPATH = $CLASSPATH%:
       $installpath/classes/ConfigManagerProxy.jar
   ```

   where:

   $installpath
   Refers to the installation directory of the product.
3. Similarly, ensure that your Java development directory is also on the CLASSPATH.

Eclipse environment

To configure the CLASSPATH:

1. Select `File -> New -> Project -> Java -> Java Project`
   a. Click `Next`
2. Enter the project name
   a. Click `Next`
3. In the `Libraries` tab, click `Add External Jars...`
4. Navigate to, and add the WebSphere MQ JARs to the build path:
   - `com.ibm.mq.jar`
   - `jms.jar`
   - `jta.jar`
   - `connector.jar`

   a. Import `mqii.properties` into your project in order to suppress the ″Message Catalog not found″ error that the WebSphere MQ classes generate. If you want to connect to a Configuration Manager running on the local machine *and* want to use the WebSphere MQ Java Bindings to do this, you also need to add the bindings shared library (for example `mqjbnd02.dll` or `wmqjbnd.so`) to your project.
5. Navigate to and add the Configuration Manager Proxy (CMP) JAR to the build path (`ConfigManagerProxy.jar`)
6. Select `OK`.

Environments without the broker component installed

You can run a set of Java applications that use the Configuration Manager Proxy (CMP applications) in environments that do not have the broker component installed. This set of CMP applications includes user-written applications and the following command utilities:

- mqsicreateexecutiongroup
- mqsideleteexecutiongroup
- mqsistartmsgflow
- mqsistopmsgflow
- mqsideploy

  WebSphere Message Broker for Windows provides two variants of the mqsideploy command, **mqsideploy.bat** and **mqsideploy.exe**. Only **mqsideploy.bat** can be used in environments that do not have the broker component installed.

To install CMP applications in an environment that does not have the broker component installed, perform the following steps:

1. Ensure that the computer that does not have the broker component installed, the target computer, has a 1.4.2-compatible Java Runtime Environment.
2. Copy the following set of files from a computer that has the broker component installed to the target computer:
   a. ConfigManagerProxy.jar from the classes directory.
   b. The WebSphere MQ Classes for Java.
      - On Windows these are located in com.ibm.mq.jar.

- On other platforms these are located in the component's installation image.

c. Your CMP application and any configuration files, for example *.configmgr files.

d. If you want to run any of the available broker utilities on the target computer, perform the following steps:

   1) Copy ConfigUtil.jar from the classes directory.

   2) Copy the required utility bat files, or shellscripts, from the bin directory. Copy one or more of the following bat files:

      - **mqsicreateexecutiongroup.bat**
      - **mqsideleteexecutiongroup.bat**
      - **mqsistartmsgflow.bat**
      - **mqsistopmsgflow.bat**
      - **mqsideploy.bat**

e. If you want to display broker (BIP) messages in non-English environments, copy all BIPv600*.properties files from the messages directory.

3. On the target computer, update the CLASSPATH environment variable to include the following files:
   - The CMP classes, ConfigManagerProxy.jar
   - The user-supplied applications that import the CMP classes
   - The WebSphere MQ Classes for Java, com.ibm.mq.jar, and any additional JARs required by this package.
   - Any other required JARs and directories. For example, if you require any of the available command utilities on the target computer include ConfigUtil.jar, or if you require the broker (BIP) messages to be displayed in non-English environments include a directory containing BIPv600*.properties.
4. Ensure that the user ID that the target computer uses has the following authorities:
   - Authority to connect to the queue manager that the Configuration Manager uses.
   - Authority to manipulate broker domain objects.

You can now run user-written CMP applications, and the specified command utilities, on the target computer.

Connecting to the Configuration Manager using the Configuration Manager Proxy

Before you start

Before starting this step, you must have completed "Configuring an environment for developing and running Configuration Manager Proxy applications" on page 11.

Consider the following program `ConnectToConfigManager.java`; it attempts to connect to a Configuration Manager running on the default queue manager of the local machine.

```
import com.ibm.broker.config.proxy.*;

public class ConfigManagerRunStateChecker {
```

```
    public static void main(String[] args) {
        displayConfigManagerRunState("localhost", 1414, "");
    }

    public static void displayConfigManagerRunState(String hostname,
                                                    int port,
                                                    String qmgr) {
        ConfigManagerProxy cmp = null;
        try {
            ConfigManagerConnectionParameters cmcp =
                new MQConfigManagerConnectionParameters(hostname, port, qmgr);
            cmp = ConfigManagerProxy.getInstance(cmcp);
            String configManagerName = cmp.getName();

            System.out.println("Configuration Manager '"+configManagerName+
                "' is available!");
            cmp.disconnect();
        } catch (ConfigManagerProxyException ex) {
            System.out.println("Configuration Manager is NOT available"+
                " because "+ex);
        }
    }
}
```

The first line of the program requests Java to import the CMP classes. All CMP classes are in the `com.ibm.broker.config.proxy` package.

The first line inside the try block of the `displayConfigManagerRunState()` method instantiates a `ConfigManagerConnectionParameters` object. This is an interface which states that implementing classes are able to provide the parameters to connect to a Configuration Manager.

The only class that implements this interface is `MQConfigManagerConnectionParameters`, which defines a set of WebSphere MQ-based connection parameters. The constructor used here takes three parameters:

1. The host name of the Configuration Manager machine
2. The port on which the Configuration Manager's WebSphere MQ listener service is listening
3. the name of the Configuration Manager's WebSphere MQ queue manager

Once you have defined this object, an attempt can be made to connect to the Configuration Manager's queue manager with those characteristics. This is achieved by the static `getInstance()` factory method just inside the `try` block. Once a valid handle to the Configuration Manager is obtained, the application attempts to discover the name of the Configuration Manager (`cmp.getName()`) and display it.

Note: `getName()` - and other methods that request information from the Configuration Manager - block until the information is supplied, or a timeout occurs.

This means that if the Configuration Manager is not running, the application hangs for a period. It is possible to control the timeout period by using the `ConfigManagerProxy.setRetryCharacteristics()` method. Generally, however, blocking only occurs when a given resource is accessed for the first time within an application.

Finally, the `disconnect()` method is called. This method frees up resources associated with the connection in both the CMP and Configuration Manager.

Note: When a `ConfigManagerProxy` handle is first returned from the `getInstance()` method, the Configuration Manager service is not necessarily running. It is only when the application attempts to make use of the handle (by calling `getName()` in this example) that the application can be assured that a two-way connection with the Configuration Manager is active

Navigating broker domains using the Configuration Manager Proxy

Before you start

Before starting this step, you must have completed "Connecting to the Configuration Manager using the Configuration Manager Proxy" on page 14.

Each domain object that is controllable from the Configuration Manager is represented as a single object in the Configuration Manager Proxy (CMP) and this includes:

- Brokers
- Execution groups
- Deployed message flows
- Topics
- Collectives
- Subscriptions
- Publish/Subscribe topology
- Broker event log

The CMP also handles deployed message sets, although these are handled as attributes of deployed execution groups.

Collectively known as administered objects these objects provide the bulk of the interface to the Configuration Manager, and as such are fundamental to understanding the Configuration Manager Proxy API.

Each administered object is an instance of a Java class that describes the underlying type of object in the Configuration Manager. The possible Java classes follow:

Java class	Class function
TopologyProxy	Describes the pub/sub topology.
CollectiveProxy	Describes pub/sub collectives.
BrokerProxy	Describes brokers.
ExecutionGroupProxy	Describes execution groups.
MessageFlowProxy	Describes message flows that have already been deployed to execution groups; does NOT describe message flows in the Broker Application Development perspective of the toolkit.
TopicProxy	Describes topics.
TopicRootProxy	Describes the root of the topic hierarchy.
LogProxy	Describes the broker's event log for the current user.

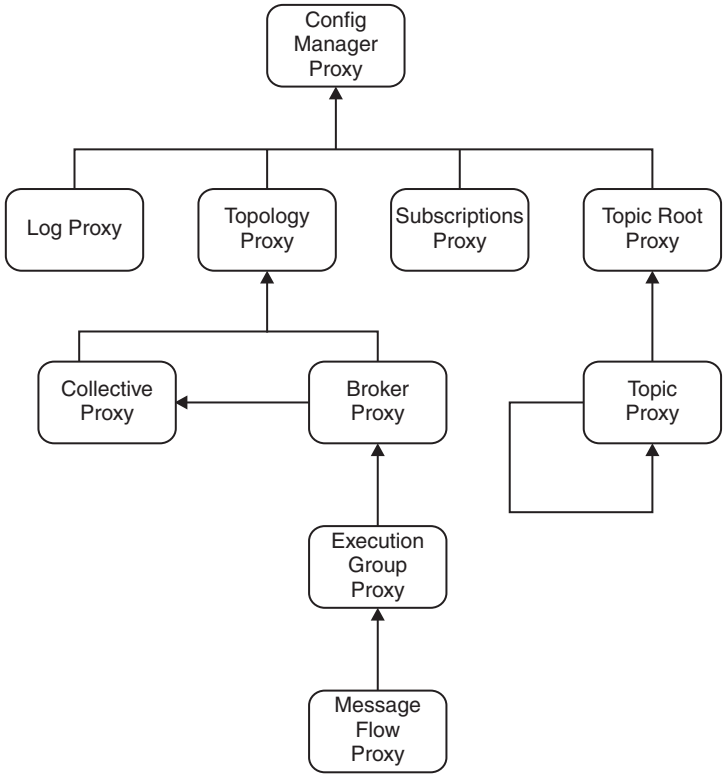
Java class	Class function
SubscriptionsProxy	Describes a subset of the active subscriptions.
ConfigManagerProxy	Describes the Configuration Manager itself.

Each administered object describes a single object that is controllable from the Configuration Manager. For example, every broker within a broker domain will have one `BrokerProxy` instance that represents it within the CMP application, and so on.

Declared in each administered object is a set of public methods that programs can use to enquire and manipulate properties of the underlying Configuration Manager object to which the instance refers. For example, on a `BrokerProxy` object that refers to broker `B1`, it is possible to invoke methods that cause the broker to reveal its run-state, or cause it to start all its message flows and so on.

To access an administered object, and make use of its API, it is necessary to first request a handle to it from the object that logically owns it. For example, as brokers logically own execution groups, in order to gain a handle to execution group `EG1` running on broker `B1` the application needs to ask the `BrokerProxy` object represented by `B1` for a handle to the `ExecutionGroupProxy` object represented by `EG1`.

In the `ConnectToConfigManager` example a handle is gained to the `ConfigManagerProxy` object. The `ConfigManagerProxy` is logically the root of the administered object tree, which means that all other objects in the Configuration Manager are directly, or indirectly, accessible from it. The Configuration Manager directly owns the Publish/Subscribe topology and so there is a method that applications can invoke from `ConfigManagerProxy` in order to gain a handle to the `TopologyProxy` object. Similarly, the topology logically contains the set of all brokers and so it is possible to call methods on the `TopologyProxy` object to access the `BrokerProxy` objects. The complete hierarchy of these access relationships is shown below:



Using the `ConnectToConfigManager` example as a starting point, the following program traverses the administered object hierarchy to discover the run state of a deployed message flow. Note that the program assumes that message flow `MF1` is deployed to `EG1` on broker `B1`, although it is possible to substitute these values in the code for any that are valid in the domain.

```
import com.ibm.broker.config.proxy.*;

public class GetMessageFlowRunState {

  public static void main(String[] args) {

    ConfigManagerProxy cmp = null;
    try {
      ConfigManagerConnectionParameters cmcp =
         new MQConfigManagerConnectionParameters(
           "localhost",
           1414,
           "");
      cmp = ConfigManagerProxy.getInstance(cmcp);
    } catch (ConfigManagerProxyException cmpex) {
      System.out.println("Error connecting: "+cmpex);
    }

    if (cmp != null) {
      System.out.println("Connected to Config Manager!");
      displayMessageFlowRunState(cmp, "B1", "EG1", "MF1");
      cmp.disconnect();
    }
  }

  private static void displayMessageFlowRunState(
                                ConfigManagerProxy cmp,
                                String brokerName,
```

```
                                String egName,
                                String flowName) {
    try {
      TopologyProxy topology = cmp.getTopology();

      if (topology != null) {
        BrokerProxy b = topology.getBrokerByName(brokerName);

        if (b != null) {
          ExecutionGroupProxy eg =
            b.getExecutionGroupByName(egName);

          if (eg != null) {
            MessageFlowProxy mf =
              eg.getMessageFlowByName(flowName);

            if (mf != null) {
              boolean isRunning = mf.isRunning();
              System.out.print("Flow "+flowName+" on " +
                egName+" on "+brokerName+" is ");

              if (isRunning) {
                System.out.println("running");
              } else {
                System.out.println("stopped");
              }
            } else {
              System.err.println("No such flow "+flowName);
            }
          } else {
            System.err.println("No such exegrp "+egName+"!");
          }
        } else {
          System.err.println("No such broker "+brokerName);
        }
      } else {
        System.err.println("Topology not available!");
      }
    } catch(ConfigManagerProxyPropertyNotInitializedException
                                                      ex) {
      System.err.println("Comms problem! "+ex);
    }
  }
}
```

The method that does most of the work is `displayMessageFlowRunState()`. This method takes the valid `ConfigManagerProxy` handle gained previously and discovers the run-state of the message flow as follows:

1. The `ConfigManagerProxy` instance is used to gain a handle to the `TopologyProxy`. As there is only ever one topology per Configuration Manager, the `getTopology()` method does not need qualifying with an identifier.
2. If a valid topology is returned, the `TopologyProxy` instance is used to gain a handle to its `BrokerProxy` object with the name described by the string `brokerName`.
3. If a valid broker is returned, the `BrokerProxy` instance is used to gain a handle to its `ExecutionGroupProxy` object with the name described by the string `egName`.
4. If a valid execution group is returned, the `ExecutionGroupProxy` instance is used to gain a handle to its `MessageFlowProxy` object with the name described by the string `flowName`.
5. If a valid message flow is returned, the run-state of the `MessageFlowProxy` object is queried and the result is displayed.

It is not necessary to know the names of objects that you intend to manipulate. Each administered object contains methods to return sets of objects that it logically owns. The following example demonstrates this by looking up the names of all brokers within the domain:

```
import java.util.Enumeration;
import com.ibm.broker.config.proxy.*;

public class DisplayBrokerNames {

  public static void main(String[] args) {

    ConfigManagerProxy cmp = null;
    try {
      ConfigManagerConnectionParameters cmcp =
         new MQConfigManagerConnectionParameters(
           "localhost",
           1414,
           "");
      cmp = ConfigManagerProxy.getInstance(cmcp);
    } catch (ConfigManagerProxyException cmpex) {
      System.out.println("Error connecting: "+cmpex);
    }

    if (cmp != null) {
      System.out.println("Connected to Config Manager!");
      displayBrokerNames(cmp);
      cmp.disconnect();
    }
  }

  private static void displayBrokerNames(ConfigManagerProxy cmp)
  {
    try {
      TopologyProxy topology = cmp.getTopology();

      if (topology != null) {
        Enumeration allBrokers = topology.getBrokers(null);

        while (allBrokers.hasMoreElements()) {
          BrokerProxy thisBroker =
            (BrokerProxy) allBrokers.nextElement();
          System.out.println("Broker "+thisBroker.getName());
        }
      }
    } catch(ConfigManagerProxyPropertyNotInitializedException
                                                        ex) {
        System.err.println("Comms problem! "+ex);
    }
  }
}
```

The key method is `TopologyProxy.getBrokers(Properties)`. When supplied with a null argument, it returns an Enumeration of all the `BrokerProxy` objects in the domain. The program uses this method to look at each `BrokerProxy` in turn and display its name.

The Properties argument of `TopologyProxy.getBrokers(Properties)` can be used to exactly specify the characteristics of the brokers that are sought. It is possible to do this for nearly all of the methods that return administered objects, and is a powerful way of filtering those objects with which the program needs to work.

Examples of those characteristics that can be used to filter object look ups are the run-state and short description, as well as more obvious properties such as the

name and UUID. In order to write logic to achieve this, it is necessary for you to understand how each administered object stores its information.

The properties of each administered object are stored locally inside the object using a hash table, where each property is represented as a {key, value} tuple. Each key is the name of an attribute (for example, name) and each value is the value (for example, BROKER1).

Each key name must be expressed using a constant from the AttributeConstants class (com.ibm.broker.config.proxy). A complete set of keys and possible values for each administered object is described in the Java documentation for the AttributesConstant class, or by using the Show raw property table for this object function in the Configuration Manager Proxy API Exerciser sample program. The latter displays the complete list of {key, value} pairs for each administered object.

The Properties argument supplied to the look up methods is a set of those {key, value} pairs that must exist in each administered object in the returned enumeration. To demonstrate this, consider the following code fragment:

```
Properties p = new Properties();

p.setProperty(AttributeConstants.OBJECT_RUNSTATE_PROPERTY,
              AttributeConstants.OBJECT_RUNSTATE_RUNNING);

Enumeration e = executionGroup.getMessageFlows(p);
```

Providing that the variable executionGroup is a valid ExecutionGroupProxy object, the returned enumeration only contains running message flows (OBJECT_RUN_STATE_PROPERTY equal to OBJECT_RUNSTATE_RUNNING).

When property filtering is applied to a method that returns a single administered object rather than an enumeration of objects, only the first result is returned (which is non deterministic if more than one match applies). This means that:

```
Properties p = new Properties();

p.setProperty(AttributeConstants.NAME_PROPERTY,
              "shares");

TopicProxy t = topicProxy.getTopic(p);
```

is an alternative to:

```
TopicProxy t = topicProxy.getTopicByName("shares");
```

If multiple {key, value} pairs are added to a property filter, all properties must be present in the child object in order for an object to match. It is not possible to perform a logical OR, or a logical NOT, on a filter without writing specific application code to do this.

When AdministeredObjects are first instantiated in an application, the CMP asks the Configuration Manager for the current set of properties for that object. This happens asynchronously, which means that the first time a property is requested there may be a pause while the CMP waits for the information to be supplied by the Configuration Manager. If the information does not arrive within a certain time (for example, if the Configuration Manager is not running), a ConfigManagerProxyPropertyNotInitializedException is thrown. The maximum time that the CMP waits is determined by the ConfigManagerProxy.setRetryCharacteristics() method.

Using the Configuration Manager Proxy API to deploy

You can use the Configuration Manager Proxy API for all possible types of deployment:

Deployment type	Description
TopologyProxy.deploy()	Deploys the publish/subscribe topology to all affected brokers.
BrokerProxy.deploy()	Deploys the broker configuration.
ExecutionGroupProxy.deploy()	Deploys a BAR file to an execution group.
TopicRootProxy.deploy()	Deploys the topic hierarchy to all brokers.
ConfigManagerProxy.cancelDeployment()	Cancels all outstanding deploys in the domain.
BrokerProxy.cancelDeployment()	Cancels any outstanding deploy to a specific broker.

The Configuration Manager Proxy API has more information about each of these methods and you can find an example of the code you might use for each type of deployment in the appropriate topic in the Deploying section.

You can also check the result of a deployment using the Configuration Manager Proxy API.

An example

Here is one example that adds a broker called *B2* that is running on queue manager *QMB2* to the domain and associates with it an execution group called 'default'. This configuration is then deployed to the broker.

For this example to work successfully, the broker *B2* has been created on the machine running queue manager *QMB2*, and it has not already been deployed to by another Configuration Manager.

```
import com.ibm.broker.config.proxy.*;

public class AddBroker {

  public static void main(String[] args) {
    ConfigManagerProxy cmp = null;
    try {
      ConfigManagerConnectionParameters cmcp =
        new MQConfigManagerConnectionParameters(
        "localhost",
        1414,
        "");
      cmp = ConfigManagerProxy.getInstance(cmcp);
    }
    catch (ConfigManagerProxyException cmpex) {
      System.out.println("Error connecting: "+cmpex);
    }
    if (cmp !=null) {
      System.out.println("Connected to Config Manager");
      addBroker(cmp, "B2", "QMB2", "default");
      cmp.disconnect();
    }
  }

  private static void addBroker(ConfigManagerProxy cmp,
```

```
                              String bName,
                              String bQMgr,
                              String egName) {
    TopologyProxy topology = null;
    try {
      topology = cmp.getTopology();
    }
    catch(ConfigManagerProxyPropertyNotInitializedException ex) {
      System.err.println("Comms problem! "+ex);
    }
    if (topology != null) {
      try {
        BrokerProxy b2 = topology.createBroker(bName, bQMgr);
        ExecutionGroupProxy e = b2.createExecutionGroup(egName);
        b2.deploy();
      }

      catch (ConfigManagerProxyException ex) {
        System.err.println("Could not perform an action: "+ex);
      }
    }
  }
}
```

Configuration Manager Proxy Exerciser

You can also use the Configuration Manager Proxy Exerciser to deploy. The exerciser is a graphical interface to the Configuration Manager Proxy that allows you to view and manipulate Configuration Manager domains. For example:

1. Connect to the Configuration Manager: **File → Connect to Configuration Manager**. This opens the Connect to Configuration Manager dialog.
2. Enter the relevant connection parameters in the dialog. A hierarchical representation of the domain is displayed.
3. You can perform a number of operations. For example:
   - Click an object in the tree to display the attributes of that object.
   - Right-click an object in the tree to invoke Configuration Manager Proxy methods that manipulate that object. For example, right-clicking a broker opens a drop-down menu that has items such as 'start user trace', 'deploy broker configuration' and 'cancel all outstanding deploys to this broker'.
   - Use the log pane at the bottom of the screen to view useful information relating to the operation being performed.

Checking the results of deployment using the Configuration Manager Proxy API

If you are using a Configuration Manager Proxy application, you can find out the result of a publish/subscribe topology deployment operation, for example, by using code similar to this:

```
TopologyProxy t = cmp.getTopology();

boolean isDelta = true;
long timeToWaitMs = 10000;
DeployResult dr = topology.deploy(isDelta, timeToWaitMs);

System.out.println("Overall result = "+dr.getCompletionCode());

// Display overall log messages
Enumeration logEntries = dr.getLogEntries();
while (logEntries.hasMoreElements()) {
```

```
  LogEntry le = (LogEntry)logEntries.nextElement();
  System.out.println("General message: " + le.getDetail());
}

// Display broker specific information
Enumeration e = dr.getDeployedBrokers();
while (e.hasMoreElements()) {

  // Discover the broker
  BrokerProxy b = (BrokerProxy)e.nextElement();

  // Completion code for broker
  System.out.println("Result for broker "+b+" = " +
    dr.getCompletionCodeForBroker(b));

  // Log entries for broker
  Enumeration e2 = dr.getLotEntriesForBroker(b);
  while (e2.hasMoreElements()) {
    LogEntry le = (LogEntry)e2.nextElement();
    System.out.println("Log message for broker " + b +
      le.getDetail()));
  }
}
```

The `deploy()` method blocks until all affected brokers have responded to the deployment request.

When the method returns, the DeployResult represents the outcome of the deployment at the time when the method returned; the object is not updated by the Configuration Manager Proxy.

If the deployment message could not be sent to the Configuration Manager, a `ConfigManagerProxyLoggedException` is thrown at the time of deployment. If the Configuration Manager receives the deployment message, then log messages for the overall deployment are displayed, followed by completion codes specific to each broker affected by the deployment. The completion code is one of the following static instances from the `com.ibm.broker.config.proxy.CompletionCodeType` class:

Completion code	**Description**
`pending`	The deploy is held in a batch and will not be sent until you issue *ConfigManagerProxy.sendUpdates()*.
`submitted`	The deploy message was sent to the Configuration Manager but no response was received before the timeout occurred.
`initiated`	The Configuration Manager replied stating that deployment has started, but no broker responses were received before the timeout occurred.
`successSoFar`	The Configuration Manager issued the deployment request and some, but not all, brokers responded with a success message before the timeout period expired. No brokers responded negatively.
`success`	The Configuration Manager issued the deployment request and all relevant brokers responded successfully before the timeout period expired.
`failure`	The Configuration Manager issued the deployment request and at least one broker responded negatively. You can use *getLogEntriesForBroker* for more information on why the deployment failed.

Completion code	Description
notRequired	A deployment request was submitted to the Configuration Manager involved with the supplied broker, but the request was not sent to the broker because its configuration is already up to date.

Managing broker domains using the Configuration Manager Proxy

Before you start

Before starting this step, you must have completed "Connecting to the Configuration Manager using the Configuration Manager Proxy" on page 14.

Using the CMP it is possible to change the state of objects in the domain– that is, create, delete, modify, and deploy objects stored within the Configuration Manager. The following example attempts to set the long description field of a broker calledB1:

```
import com.ibm.broker.config.proxy.*;

public class SetLongDescription {

  public static void main(String[] args) {

    ConfigManagerProxy cmp = null;
    try {
      ConfigManagerConnectionParameters cmcp =
         new MQConfigManagerConnectionParameters(
           "localhost",
           1414,
           "");
      cmp = ConfigManagerProxy.getInstance(cmcp);
    } catch (ConfigManagerProxyException cmpex) {
      System.out.println("Error connecting: "+cmpex);
    }

    if (cmp != null) {
      System.out.println("Connected to Config Manager!");
      describeBroker(cmp, "B1", "this is my broker");
      cmp.disconnect();
    }
  }

  private static void describeBroker(ConfigManagerProxy cmp,
                                     String brokerName,
                                     String newDesc)
  {
   BrokerProxy b = null;
    try {
      TopologyProxy topology = cmp.getTopology();
      if (topology != null) {
        b = topology.getBrokerByName(brokerName);
      }
    } catch(ConfigManagerProxyPropertyNotInitializedException
                                                      ex) {
        System.err.println("Comms problem! "+ex);
    }

    if (b != null) {
      try {
        b.setLongDescription(newDesc);
      } catch (ConfigManagerProxyException ex) {
```

```
        System.err.println("Could not send request to CM: "+ex);
      }
    } else {
      System.err.println("Broker "+brokerName+" not found");
    }
  }
}
```

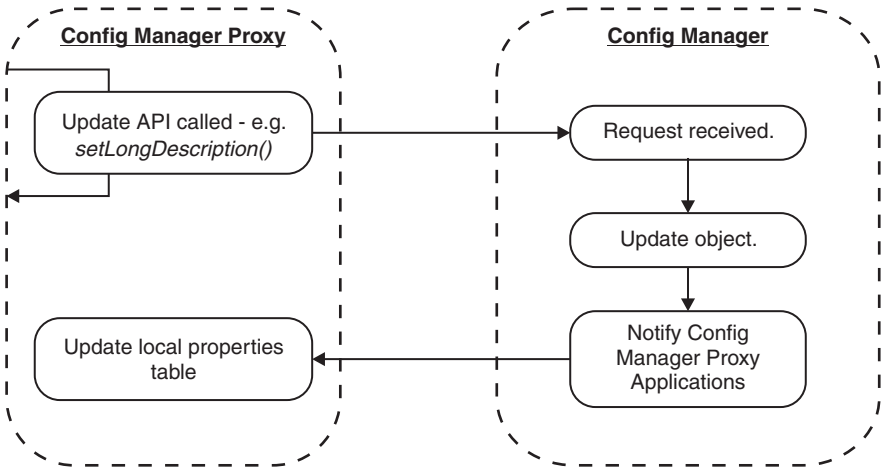
The `setLongDescription()` method works by asking the Configuration Manager to modify a (`key, value`) property of the broker `B1`, where the key name represents the long description tag, and the value is the new long description. So an alternative to calling `setLongDescription()` is:

```
Properties p = new Properties();

p.setProperty(AttributeConstants.LONG_DESCRIPTION_PROPERTY,
              newDesc);

b.setProperties(p);
```

When the request to change properties is sent to the Configuration Manager, The CMP's internal properties tables are not updated until the Configuration Manager reports that its copy of the attributes has been changed successfully. This is done in order to keep all copies of the information consistent. This process is shown below.



Note, that if the current user does not have the necessary permissions, as `SetLongDescription.java` works it is not possible to determine if the request gets rejected by the Configuration Manager. The CMP method to set the long description field throws a `ConfigManagerProxyException` if, and only if, the message to perform the operation can not be sent to the Configuration Manager. This means that output from the program is exactly the same, even if the Configuration Manager can not change the required property.

The reason for this is that the Configuration Manager processes requests from the CMP asynchronously, and so it could theoretically be a considerable time until the action is performed at the Configuration Manager. If methods such as the one described within this topic did not return control to the program until the completion codes became available, the performance of the CMP application would be wholly dependent on the performance of the Configuration Manager.

Next:

The design of most state-changing CMP methods is to return immediately without informing the calling application of the outcome of the request. To discover this information refer to "Checking the results of broker domain management using the Configuration Manager Proxy"

Checking the results of broker domain management using the Configuration Manager Proxy

This is part of the larger task of developing Configuration Manager Proxy (CMP) applications.

There are three ways of determining the outcome of requests to manipulate, that is, create, delete, modify, and deploy Configuration Manager objects:

- For deployment methods only, it is possible to use the return code from the deployment API; see "Checking the results of broker domain management using the Configuration Manager Proxy with return codes"
- By using an API to query an object's last completion code; see "Checking the results of broker domain management using the Configuration Manager Proxy with the last completion code" on page 29
- By using the administered object notification mechanism which is the recommended approach; see "Checking the results of broker domain management using the Configuration Manager Proxy with object notification" on page 30

Checking the results of broker domain management using the Configuration Manager Proxy with return codes

This is part of the larger task of developing Configuration Manager Proxy (CMP) applications.

The only state-changing methods that supply a return code representing the outcome of the request are the deploy() methods. The following sample of code shows how to discover the outcome of a topology deploy operation using the returned `DeployResult` object:

```
...

TopologyProxy t = cmp.getTopology();

boolean isDelta = true;
long timeToWaitMs = 10000;
DeployResult dr = topology.deploy(isDelta, timeToWaitMs);

System.out.println("Overall result = "+dr.getCompletionCode());

// Display overall log messages
Enumeration logEntries = dr.getLogEntries();
while (logEntries.hasMoreElements()) {
    LogEntry le = (LogEntry)logEntries.nextElement();
    System.out.println("General message: " + le.getDetail());
}

// Display broker specific information
Enumeration e = dr.getDeployedBrokers();
while (e.hasMoreElements()) {

    // Discover the broker
    BrokerProxy b = (BrokerProxy)e.nextElement();

    // Completion code for broker
```

```
    System.out.println("Result for broker "+b+" = " +
        dr.getCompletionCodeForBroker(b));

    // Log entries for broker
    Enumeration e2 = dr.getLogEntriesForBroker(b);
    while (e2.hasMoreElements()) {
        LogEntry le = (LogEntry)e2.nextElement();
        System.out.println("Log message for broker " + b +
            le.getDetail()));
    }
}
```

In this code the `deploy()` method is blocked until all affected brokers have responded to the deployment request. However, the method includes a `long` parameter that describes the maximum length of time the CMP waits for the responses to arrive.

Note that when the method finally returns, the `DeployResult` represents the outcome of the deployment at the time the method returned. In other words, once returned to the application, the object is not updated by the CMP.

After the `deploy()` method completes, the example interrogates the returned `DeployResult` and displays the overall completion code for the deploy operation. This takes one of the following values:

(com.ibm.broker.config.proxy.)CompletionCodeType.pending
: Means that the deploy is held in a batch and is not sent until you issue `ConfigManagerProxy.sendUpdates()`. Note that if this message applies it is returned immediately – that is, without waiting for the timeout period to expire.

CompletionCodeType.submitted
: Means that the deploy message was sent to the Configuration Manager but no response was received before the timeout occurred. Note that if the deployment message can not be sent to the Configuration Manager, a `ConfigManagerProxyLoggedException` is thrown at deploy time instead.

CompletionCodeType.initiated
: Means that the Configuration Manager replied stating that deployment has started, but no broker responses were received before the timeout occurred.

CompletionCodeType.successSoFar
: Means that the Configuration Manager issued the deployment request and some, but not all, brokers responded with a ″success″ message before the timeout period expired. No brokers responded negatively.

CompletionCodeType.success
: Means that the Configuration Manager issued the deployment request, and all relevant brokers responded successfully before the timeout period expired. This message is sent as soon as all relevant brokers have responded successfully.

CompletionCodeType.failure
: Means that the Configuration Manager issued the deployment request, and at least one broker responded negatively.

Note, that not all completion codes apply to all deploys. For example, deploying to a single specific broker cannot result in a completion code of `'successSoFar'`.

The example next displays any log messages from the deployment that can not be attributed to any specific broker. On a successful deploy, these messages always

include a ″deploy initiated″ log entry originating from the Configuration Manager, even if the deployment subsequently completed.

Finally, the example displays the completion code and any log messages specific to each broker affected by the deployment. Note that on a topology or topic tree deploy, this is every broker in the domain.

The set of completion codes applicable to a response from a specific broker are:

CompletionCodeType.pending
: Means that the deploy is held in a batch and is not sent until you issue `ConfigManagerProxy.sendUpdates()`.

CompletionCodeType.submitted
: Means that the deploy message was sent but no response has yet been received from the Configuration Manager stating that deployment has been initiated.

CompletionCodeType.initiated
: Means that the Configuration Manager has replied, stating that deployment has started, but no reply has yet been returned from the broker.

CompletionCodeType.success
: Means that the Configuration Manager issued the deployment request, and the broker successfully applied the deployment changes.

CompletionCodeType.failure
: Means that the Configuration Manager issued the deployment request, and the broker responded by stating that the deployment was not successful. Use `getLogEntriesForBroker()` for more information on why the deployment failed.

CompletionCodeType.notRequired
: Means that a deployment request was submitted to the Configuration Manager that involved the supplied broker, but the broker was not sent the request because its configuration is already up to date.

See “Running the Deploy BAR sample” on page 5 or “Running the broker domain management sample” on page 6 `CMPAPIExerciser.reportDeployResult()` method for examples of how to parse `DeployResult` objects.

Checking the results of broker domain management using the Configuration Manager Proxy with the last completion code

This is part of the larger task of developing Configuration Manager Proxy (CMP) applications.

Most state-changing methods in the CMP do not make use of the return code in this way. For such methods, discovering the outcome of an action can be slightly more complicated. Assuming that administered objects are not shared across threads, the following code fragment can be used to discover the outcome of a request to modify a broker’s `LongDescription`, where `b` is an instance of `BrokerProxy`:

```
GregorianCalendar oldCCTime =
                          b.getTimeOfLastCompletionCode();
b.setLongDescription(newDesc);
GregorianCalendar newCCTime = oldCCTime;
while (oldCCTime.equals(newCCTime)) {
  newCCTime = b.getTimeOfLastCompletionCode());
```

```
  Thread.sleep(1000);
}
CompletionCodeType ccType = b.getLastCompletionCode();
if (ccType == CompletionCodeType.success) {
  // etc.
}
```

This example causes the application to continually query the time the topology last received a completion code; that is, when an action on the topology was last completed. When the results of the `createBroker()` are returned to the CMP, the completion code is updated and control breaks out of the `while` loop. At this point the last completion code is determined.

As well as being unsuitable for a multi-threaded application, this algorithm for determining the outcome of commands is inefficient as it causes the CMP application to wait while the Configuration Manager processes the request.

A better way of doing this is to make use of administered object notifications; see "Checking the results of broker domain management using the Configuration Manager Proxy with object notification."

Checking the results of broker domain management using the Configuration Manager Proxy with object notification

This is part of the larger task of developing Configuration Manager Proxy (CMP) applications.

It is possible to notify applications whenever commands complete, or whenever changes occur to administered objects. By making use of the `OBSERVER` design pattern, it is possible to supply the CMP with a handle to a user-supplied object that has a specific method invoked if an object is modified, deleted, or whenever a response to a previously submitted action is returned from the Configuration Manager.

The user-supplied code must implement the `AdministeredObjectListener` interface. It defines methods that are invoked by the CMP when an event occurs on an administered object to which the listener is registered. These methods are:

- `processModify(...)`
- `processDelete(...)`
- `processActionResponse(...)`

`processModify(...)` is invoked whenever the administered object to which the listener is registered has one or more of its attributes modified by the Configuration Manager. Information supplied on this notification, through the use of the `processModify()` method arguments are a:

1. Handle to the `AdministeredObject` to which the notification refers.
2. List of strings containing the key names that have been changed.
3. List of strings describing any new subcomponents that have just been created for the object, for example, new execution groups in a broker.
4. List of strings describing any subcomponents that have just been removed for the object.

The format of the strings passed to the final two parameters is an internal representation of the administered object. It is possible to turn this representation into an administered object type by using the `getSubcomponentFromString()` method.

Note:

1. Strings are passed within these lists to enhance performance; the CMP does not use resource instantiating administered objects unless they are specifically requested by the calling application.
2. The first time you call the `processModify()` method for a listener, the `changed attributes` parameter can include a complete set of attribute names for the object, if the application is using a batch method, or if the CMP is experiencing communication problems with the Configuration Manager.

`processDelete(...)` is invoked if the object with which the listener is registered is completely removed from the Configuration Manager. Supplied to `processDelete(...)` is one parameter – a handle to the administered object that has been deleted; once this method returns, the administered object handle might no longer be valid. Around the same time that a `processDelete(...)` event occurs, a `processModify(...)` event is sent to listeners of the deleted object's parent, to announce a change in the parent's list of subcomponents.

`processActionResponse(...)` is the event that informs the application that a previous action submitted by that application is complete, and there is only one `processActionResponse(...)` event received for each state-changing operation issued by the CMP application. Supplied to this event are the following pieces of information:

1. A handle to the administered object for which a request was submitted.
2. The completion code of the request.
3. A set of zero, or more, informational (BIP) messages associated with the result.
4. A set of (`key, value`) pairs that describes the submitted request in more detail.

Consult the *Configuration Manager Proxy API Reference* for information on parsing the pairs in the last parameter.

In order to register a listener, each administered object has a `registerListener()` method that is used to tell the CMP to call the supplied code whenever an event occurs on that object. It is possible to register the same `AdministeredObjectListener` for notifications from multiple administered objects. In addition, it is possible to register multiple `AdministeredObjectListeners` against the same administered object.

The following example demonstrates this by registering a listener on the topology object and displaying a message whenever it is modified:

```
import com.ibm.broker.config.proxy.*;
import com.ibm.broker.config.common.CompletionCodeType;
import java.util.List;
import java.util.ListIterator;
import java.util.Properties;

public class MonitorTopology implements AdministeredObjectListener {

  public static void main(String[] args) {

    ConfigManagerProxy cmp = null;
```

```
  try {
    ConfigManagerConnectionParameters cmcp =
       new MQConfigManagerConnectionParameters(
         "localhost",
         1414,
         "");
    cmp = ConfigManagerProxy.getInstance(cmcp);
  } catch (ConfigManagerProxyException cmpex) {
    System.out.println("Error connecting: "+cmpex);
  }

  if (cmp != null) {
    System.out.println("Connected to Config Manager!");
    TopologyProxy topology = cmp.getTopology();
    listenForChanges(topology);
    cmp.disconnect();
  }
}

private static void listenForChanges(AdministeredObject obj)
{
  try {
    if (obj != null) {
     obj.registerListener(new MonitorTopology());
     while(true) {
       // thread could do something else here instead
       try {
         Thread.sleep(10000);
       } catch (InterruptedException ex) {
         // ignore
       }
      }
    }
  } catch(ConfigManagerProxyPropertyNotInitializedException
                                                        ex) {
      System.err.println("Comms problem! "+ex);
  }
}

public void processActionResponse(AdministeredObject obj,
                                  CompletionCodeType cc,
                                  List bipMessages,
                                  Properties refProperties) {
  // Event ignored in this example
}

public void processDelete(AdministeredObject deletedObject) {
 // Event ignored in this example
}

public void processModify(AdministeredObject affectedObject,
                          List changedAttributes,
                          List newChildren,
                          List removedChildren) {

  System.out.println(affectedObject+" has changed:");
  ListIterator e = changedAttributes.listIterator();
  while (e.hasNext()) {
    String changedAttribute = (String) e.next();
    System.out.println("Changed: "+changedAttribute);
  }
  ListIterator e2 = newChildren.listIterator();
  while (e2.hasNext()) {
    String newChildStr = (String) e2.next();
    AdministeredObject newChild =
      affectedObject.getSubcomponentFromString(newChildStr);
    System.out.println("New child: "+newChild);
```

```
    }
    ListIterator e3 = removedChildren.listIterator();
    while (e3.hasNext()) {
      String remChildStr = (String) e3.next();
      AdministeredObject removedChild =
        affectedObject.getSubcomponentFromString(remChildStr);
      System.out.println("Removed child: "+removedChild);
    }

  }
}
```

The `listenForChanges()` method attempts to register an instance of the `MonitorTopology` class for notifications of topology changes. If successful, the main thread pauses indefinitely to prevent the application from exiting once the method returns. Once the listener is registered, whenever the topology changes - for example, if a broker is added - the `processModify()` method is called. This displays details of each notification on the screen.

There are three ways to stop receiving notifications:

- `AdministeredObject.deregisterListener(AdministeredObjectListener)`
- `ConfigManagerProxy.deregisterListeners()`
- `ConfigManagerProxy.disconnect()`

The first method de-registers a single listener from a single administered object; the other two methods deregister all listeners connected with that `ConfigManagerProxy` instance. In addition, the final method shows that all listeners are implicitly removed when connection to the Configuration Manager is stopped.

Note: You can also implement the `AdvancedAdministeredObjectListener` interface which, when registered, yields additional information to applications.

Creating domain objects using the Configuration Manager Proxy

This is part of the larger task of developing Configuration Manager Proxy (CMP) applications.

The following example adds a broker called `B2`, that is running on queue manager `QMB2`, to the domain and associates with it an execution group called `default`. Finally, this configuration is deployed to the broker.

In order for this example to work successfully, the broker `B2` must already exist on the machine running queue manager `QMB2`, and another Configuration Manager must not have deployed to it previously.

```
import com.ibm.broker.config.proxy.*;

public class AddBroker {

  public static void main(String[] args) {

    ConfigManagerProxy cmp = null;
    try {
      ConfigManagerConnectionParameters cmcp =
         new MQConfigManagerConnectionParameters(
           "localhost",
           1414,
           "");
```

```
      cmp = ConfigManagerProxy.getInstance(cmcp);
    } catch (ConfigManagerProxyException cmpex) {
      System.out.println("Error connecting: "+cmpex);
    }

    if (cmp != null) {
      System.out.println("Connected to Config Manager!");
      addBroker(cmp, "B2", "QMB2", "default");
      cmp.disconnect();
    }
  }
  private static void addBroker(ConfigManagerProxy cmp,
                                String bName,
                                String bQMgr,
                                String egName)
  {
   TopologyProxy topology = null;
    try {
      topology = cmp.getTopology();
    } catch(ConfigManagerProxyPropertyNotInitializedException
                                                        ex) {
        System.err.println("Comms problem! "+ex);
    }

    if (topology != null) {
      try {
        BrokerProxy b2 = topology.createBroker(bName, bQMgr);
        ExecutionGroupProxy e = b2.createExecutionGroup(egName);
        b2.deploy();
      } catch (ConfigManagerProxyException ex) {
        System.err.println("Could not perform an action: "+ex);
      }
    }
  }
}
```

The critical statements in this example are the three lines inside the `try` block towards the end of the `addBroker()` method. The first statement attempts to add the broker to the Configuration Manager's topology, the second attempts to create the default execution group, and the third attempts to deploy the configuration (that is, the new execution group) to the broker.

Note that the `createBroker()` method assumes that the "physical" broker component has already been created using the mqsicreatebroker command.

Because requests are processed asynchronously by the Configuration Manager, the `BrokerProxy` object that is returned from the `createBroker()` method is a "skeleton" object when returned to your application, as it refers to an object that may not yet exist on the Configuration Manager.

This is also true of the `ExecutionGroupProxy` object `e` returned from the `createExecutionGroup()` method. In both cases, the object can be manipulated by the application as if it existed on the Configuration Manager, although the actual creation of the underlying object might not happen for some time.

Once the object represented by the skeleton is created in the Configuration Manager, any requests that refer to it can be processed. In this example, once the broker has actually been added to the topology in the Configuration Manager, the Configuration Manager can honor the request to create the execution group.

If, for any reason the request to create the object described by the skeleton fails, any requests that use the skeleton also fails. So, if broker `B2` can not be created, any

requests involving the skeleton BrokerProxy object b2, that is, b2.createExecutionGroup() and b2.deploy() also fail. However, the CMP application works, as in the successful case, as no exception is thrown. See "Checking the results of broker domain management using the Configuration Manager Proxy" on page 27 for further information on how to detect problems such as these.

Advanced features of the Configuration Manager Proxy

This is part of the larger task of developing Configuration Manager Proxy (CMP) applications and introduces the advanced features of the CMP.

Follow the link for the advanced feature that you require:

- "The Configuration Manager Proxy subscriptions API"
- "Submitting batch requests using the Configuration Manager Proxy" on page 37

The Configuration Manager Proxy subscriptions API

This is part of the larger task of developing Configuration Manager Proxy (CMP) applications and is one of the advanced features of the CMP.

You can use the CMP to show and delete the set of active subscriptions in the domain. The following example gives information on all subscriptions to topics with names that begin with the string ″shares″.

```
import java.util.Enumeration;
import com.ibm.broker.config.proxy.*;

public class QuerySubscriptions {

  public static void main(String[] args) {

    ConfigManagerProxy cmp = null;
    try {
      ConfigManagerConnectionParameters cmcp =
         new MQConfigManagerConnectionParameters(
           "localhost",
           1414,
           "");
      cmp = ConfigManagerProxy.getInstance(cmcp);
    } catch (ConfigManagerProxyException cmpex) {
      System.out.println("Error connecting: "+cmpex);
    }

    if (cmp != null) {
      System.out.println("Connected to Config Manager!");
      querySubscriptionsByTopic(cmp, "shares%");
      cmp.disconnect();
    }
  }

  private static void querySubscriptionsByTopic(
    ConfigManagerProxy cmp,
    String topic)
  {
    try {
      SubscriptionsProxy matchingSubscriptions =
        cmp.getSubscriptions(topic, // filter by topic
                             null, // don't filter by broker
                             null, // don't filter by username
                             null, // don't filter by sub point
```

```
                              null, // no start date,
                              null); // no end date

        Enumeration e = matchingSubscriptions.elements();
        int matches = matchingSubscriptions.getSize();
        System.out.println("Found "+matches+" matches:");

        while (e.hasMoreElements()) {
          Subscription thisSub = (Subscription)e.nextElement();
          System.out.println("-----");
          System.out.println("Broker="+thisSub.getBroker());
          System.out.println("Topic="+thisSub.getTopicName());
          System.out.println("Client="+thisSub.getClient());
          System.out.println("Filter="+thisSub.getFilter());
          System.out.println("Reg date="
            +thisSub.getRegistrationDate());
          System.out.println("User="+thisSub.getUser());
          System.out.println("Sub point="
            +thisSub.getSubscriptionPoint());
        }
      } catch (ConfigManagerProxyException e) {
        e.printStackTrace();
      }
    }
  }
```

The method that queries the set of active subscriptions is `ConfigManagerProxy.getSubscriptions()`, which defines the query used to filter the subscriptions. The topic, broker, user ID, and `subscriptionPoint` parameters are strings which can include the `%` character to denote wild card characters.

The `startDate` and `endDate` parameters are of type `GregorianCalendar`, which can be used to constrain the registration time of the matching subscriptions. For all parameters to this method, a null value or, in the case of the string arguments, an empty value means ″do not filter by this attribute″.

In the preceding example, the only non-null parameter that is supplied to this method is the `topic` string `shares%`, that tells the CMP to return all subscriptions whose topic name begins with ″shares″.

Returned from this method is an instance of `SubscriptionsProxy` which represents the results of the query. As this class inherits from `AdministeredObject`, the attributes of this object are supplied asynchronously from the Configuration Manager and so the methods that interrogate its attributes can block temporarily while the CMP waits for the information to arrive.

Note that the `Subscription` object, which represents an individual match from the query, is a small data structure used for convenience by the `SubscriptionsProxy` and as such does not block or throw exceptions.

Despite being of `AdministeredObject` type, *SubscriptionsProxy objects cannot have AdministeredObjectListeners* registered against them. This means that once the results of a query are returned from the Configuration Manager, you are not notified if the set of matching subscriptions changes, unless you resubmit the query. The consequence of this behavior is that the results of subscriptions queries are guaranteed correct only at the time the original query was made.

It is possible to delete subscriptions using the `SubscriptionsProxy.deleteSubscriptions()` method. As `SubscriptionsProxy`

objects cannot have AdministeredObjectListeners, the outcome of such an action is published to listeners of the ConfigManagerProxy object.

Submitting batch requests using the Configuration Manager Proxy

This is part of the larger task of developing Configuration Manager Proxy (CMP) applications and is one of the advanced features of the CMP.

Using the CMP it is possible to group multiple requests destined for the same Configuration Manager together and submit them as a single unit of work.

To start a batch the application must call the beginUpdates() method on the ConfigManagerProxy handle. This tells the CMP to hold back from submitting any state-changing requests to the Configuration Manager until it is told otherwise. The sendUpdates() method tells the CMP to submit as a batch any requests received since the last beginUpdates() call, and clearUpdates() can be used to discard a batch without submitting it to the Configuration Manager. It is possible to determine whether a batch is currently in progress by using the isBatching() method. Note that there can only be one batch in progress for each CMP handle.

One advantage of using a batch method is that it provides an assurance that no other applications can have messages processed by the Configuration Manager during the batch. When a Configuration Manager receives a batch of requests, it processes each request in the batch in the order it was added to the batch (FIFO), and requests from no other CMP application are processed until the entire batch is completed.

To illustrate this, consider the following sequence of commands:

```
BrokerProxy b2 = topology.createBroker("B2", "QMB2");
ExecutionGroupProxy e = b2.createExecutionGroup("default");
b2.deploy();
```

Without using a batch method it is not possible to guarantee the success of these actions. For example, even if each command would otherwise succeed, it is possible for a second (possibly remote) application to delete the broker B2 after it has been created by the first application, but before the other two commands are processed.

If the sequence is extended to use a batch method, the Configuration Manager is now guaranteed to process all the commands together, meaning that no other application can disturb the logic intended by the application.

```
cmp.startUpdates();
BrokerProxy b2 = topology.createBroker("B2", "QMB2");
ExecutionGroupProxy e = b2.createExecutionGroup("default");
b2.deploy();
cmp.sendUpdates();
```

Another advantage of using a batch method is performance. The CMP typically sends one WebSphere MQ message to the Configuration Manager for each request. In a situation that requires lots of requests to be sent in quick succession – the creation of a topic hierarchy, for example, a batch method has a significant impact on performance in terms of both time taken to process the requests and memory. Each batch of requests is sent in a single WebSphere MQ message and so the overhead for each method is drastically reduced.

Batch mode does not provide transactional (commit and backout) capability; it is possible that some requests in a batch succeed and others fail. If the Configuration Manager processes a request in a batch that fails, it continues to process the next request in the batch regardless.

Part 2. Appendixes

Appendix. Notices

This information was developed for products and services offered in the U.S.A.

IBM may not offer the products, services, or features discussed in this information in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this information. The furnishing of this information does not give you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
U.S.A.

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

IBM World Trade Asia Corporation
Licensing
2-31 Roppongi 3-chome, Minato-ku
Tokyo 106-0032,
Japan

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law:

INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the information. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this information at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

IBM United Kingdom Laboratories,
Mail Point 151,
Hursley Park,
Winchester,
Hampshire,
England
SO21 2JN

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this information and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Programming License Agreement, or any equivalent agreement between us.

Any performance data contained herein was determined in a controlled environment. Therefore, the results obtained in other operating environments may vary significantly. Some measurements may have been made on development-level systems and there is no guarantee that these measurements will be the same on generally available systems. Furthermore, some measurements may have been estimated through extrapolation. Actual results may vary. Users of this document should verify the applicable data for their specific environment.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

All statements regarding IBM's future direction or intent are subject to change or withdrawal without notice, and represent goals and objectives only.

This information includes examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

COPYRIGHT LICENSE:

This information contains sample application programs in source language, which illustrate programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not

been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs.

Each copy or any portion of these sample programs or any derivative work, must include a copyright notice as follows:

(C) (*your company name*) (*year*). Portions of this code are derived from IBM Corp. Sample Programs. © Copyright IBM Corp. *_enter the year or years_*. All rights reserved.

Trademarks

The following terms are trademarks of International Business Machines Corporation in the United States, other countries, or both:

AIX	CICS	Cloudscape
DB2	DB2 Connect	DB2 Universal Database
developerWorks	Domino	
Everyplace	FFST	First Failure Support Technology
IBM	IBMLink	IMS
IMS/ESA	iSeries	Language Environment
Lotus	MQSeries	MVS
NetView	OS/400	OS/390
pSeries	RACF	Rational
Redbooks	RETAIN	RS/6000
SupportPac	Tivoli	VisualAge
WebSphere	xSeries	z/OS
zSeries		

Java and all Java-based trademarks and logos are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

Microsoft, Windows, Windows NT, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both.

Intel and Pentium are trademarks or registered trademark of Intel Corporation or its subsidiaries in the United States and other countries.

UNIX is a registered trademark of The Open Group in the United States and other countries.

Linux is a trademark of Linus Torvalds in the United States, other countries, or both.

Other company, product, or service names may be trademarks or service marks of others.

Index

C

D

T

Printed in USA