Java Dynamic Management Kit 5.1
Tools Reference Guide

Sun Microsystems, Inc.
4150 Network Circle
Santa Clara, CA 95054
U.S.A.

Part No: 816–7608–10
June, 2004

040519@8606

Contents

Preface

The Java™ Dynamic Management Kit (Java DMK) 5.1 provides a set of Java classes and tools for developing dynamic management solutions. This product conforms to the Java Management Extensions (JMX), v1.2 Maintenance Release, and the JMX Remote API, v1.0. These specifications define a three-level architecture:

- Instrumentation of resources
- Dynamic agents
- Remote management applications

The JMX architecture is applicable to network management, remote system maintenance, application provisioning, and the management needs of the service-based network.

The *Java Dynamic Management Kit 5.1 Tools Reference Guide* presents the development tools provided with Java DMK. This book covers the following topics.

- The `mibgen` tool for generating MBeans and relevant classes from SNMP MIBs.
- The output of the HTML protocol adaptor.
- Legacy `proxygen` tool for generating manager-side proxy objects.
- Tracing and debugging mechanism.

These tools can help you to develop management solutions to suit your requirements.

Who Should Use This Book

This book is aimed at developers who want to use the tools provided with Java DMK 5.1.

You should be familiar with Java programming, the JavaBeans™ component model, and the latest versions of the JMX and JMX Remote API specifications.

This book is not intended to be an exhaustive reference. For more information about each of the management levels, see the *Java Dynamic Management Kit 5.1 Tutorial*, and the API documentation generated by the Javadoc™ tool and included in the online documentation package.

Before You Read This Book

To use the tool commands described in this book, you must have a complete installation of the Java Dynamic Management Kit 5.1 on your system. For information about hardware and software requirements, how to install the product components and how to configure your environment, see the *Java Dynamic Management Kit 5.1 Installation README*.

Related Documentation

The Java DMK documentation set includes the following documents:

Book Title	Part Number
Java Dynamic Management Kit 5.1 Installation README	N/A
Java Dynamic Management Kit 5.1 Getting Started Guide	816–7607
Java Dynamic Management Kit 5.1 Tutorial	816–7609
Java Dynamic Management Kit 5.1 Tools Reference Guide	816–7608
Java Dynamic Management Kit 5.1 Release Notes	N/A

These books are available online after you have installed the Java DMK documentation package. The online documentation also includes the API documentation generated by the Javadoc tool for the Java packages and classes. To access the online documentation, using any web browser, open the home page corresponding to your platform.

Operating Environment	Homepage Location
Solaris / Linux / Windows 2000	*installDir*`/SUNWjdmk/5.1/doc/index.html`

In these file names, *installDir* refers to the base directory or folder of your Java DMK installation. In a default installation procedure, *installDir* is as follows.

- `/opt` on the Solaris or Linux platforms
- `C:\Program Files` on the Windows 2000 platform

These conventions are used throughout this book whenever referring to files or directories that are part of the installation.

The Java Dynamic Management Kit relies on the management architecture of two Java Specification Requests (JSRs): the JMX specification (JSR 3) and the JMX Remote API specification (JSR 160). The specification documents and reference implementations of these JSRs are available at:

`http://java.sun.com/products/JavaManagement/download.html`

How This Book Is Organized

This book describes the development tools provided with the Java DMK 5.1and explains how to use them. It is divided into the following chapters:

- Chapter 1: “SNMP MIB Compiler (`mibgen`)”
- Chapter 2: “HTML Protocol Adaptor”
- Chapter 3: “Proxy MBean Compiler (`proxygen`)”
- Chapter 4: “Tracing Mechanism”

Accessing Sun Documentation Online

The docs.sun.com^{SM} Web site enables you to access Sun technical documentation online. You can browse the docs.sun.com archive or search for a specific book title or subject. The URL is `http://docs.sun.com`.

Ordering Sun Documentation

Sun Microsystems offers select product documentation in print. For a list of documents and how to order them, see "Buy printed documentation" at `http://docs.sun.com`.

Typographic Conventions

The following table describes the typographic conventions used in this book.

Typeface or Symbol	Meaning	Example
`AaBbCc123`	The names of commands, files, and directories; on-screen computer output	Edit your `.login` file. Use `ls -a` to list all files. `machine_name% you have mail.`
`AaBbCc123`	What you type, contrasted with on-screen computer output	`machine-name%` **`su`** `Password:`
AaBbCc123	Command-line placeholder: replace with a real name or value	To delete a file, type **`rm`** *filename*.
AaBbCc123	Book titles, new words, or terms, or words to be emphasized.	Read Chapter 6 in *User's Guide*. These are called *class* options. You must be *root* to do this.

Shell Prompts in Command Examples

The following table shows the default system prompt and superuser prompt for the C shell, Bourne shell, and Korn shell.

Shell	Prompt
C shell prompt	`machine-name%`
C shell superuser prompt	`machine-name#`
Bourne shell and Korn shell prompt	`$`
Bourne shell and Korn shell superuser prompt	`#`

CHAPTER 1

SNMP MIB Compiler (`mibgen`)

The Java Dynamic Management Kit (Java DMK) provides a toolkit for developing SNMP agents and managers. This toolkit includes the SNMP MIB Compiler, `mibgen`, which is used for compiling SNMP MIBs into Java source code for agents and managers.

This chapter describes how to use the `mibgen` compiler, in the following sections:

1.1 Overview of the `mibgen` Compiler

The `mibgen` tool is a Java technology-based SNMP MIB compiler that takes an SNMP MIB as input and generates a set of Managed Beans (MBeans). These MBeans can be customized to implement the MIBs, enabling the Java DMK agent to be managed by an SNMP manager. You can use standard MBeans, model MBeans, dynamic MBeans and open MBeans in conjunction with the `mibgen` compiler.

The `mibgen` compiler is able to process the following:

- Tables with cross–references that are indexed across several MIBs
- Nested groups
- Default value variables
- Row status, namely tables controlled by a columnar object obeying the `RowStatus` convention, as defined in RFC 2579.

1.2 Starting the `mibgen` Compiler

To start the `mibgen` compiler, type the following command.

`prompt%` *installDir*`/SUNWjdmk/5.1/bin/mibgen` [*options*] *mib1 ... mibN*

Note – The script for starting the `mibgen` tool uses the `JAVA_HOME` environment variable to determine the path to the Java 2 Platform, Standard Edition (J2SE). Therefore, even if you have the correct path to the J2SE platform in your `PATH` environment variable, this is overwritten by the `JAVA_HOME` variable.

1.3 `mibgen` Options

To invoke the `java.com.sun.jdmk.tools.MibGen` class, you need to invoke `java.com.sun.jdmk.tools.MibGen` *options mib files*.

`mibgen` *options mib files*

The *options* are listed below.

`-n`	Parses the MIB files without generating code.
`-d` *dir*	Generates code in the specified target directory.
`-tp` *pkgName*	Generates code within the specified Java package (target package).
`-desc`	Includes the `DESCRIPTION` clause of `OBJECT-TYPE` as comment in generated code.
`-mo`	(*m*anager-*o*nly) Generates code for the SNMP manager only, namely the metadata file for the MIB variables (`SnmpOidTable` file). By default, the `mibgen` compiler generates code for both SNMP agents and managers. By selecting the `-mo` option, you enable the `mibgen` compiler to generate code for only the manager and not for agents. The `-mo` option is incompatible with the `-n` option.
`-mc`	(*M*IB-CORE) Does not use the default `MIB-CORE` definitions file provided with Java DMK. In this case, the user must specify the `MIB-CORE` definitions file as one of the MIB files. For example, `java.com.sun.jdmk.tools.MibGen -mc` *mib my_mib_core*.

`-a`	Generates code for all the MIB files. Without this option, the Java code is generated only for the first MIB file. In this case, the following MIB files are used to resolve some definitions of the first MIB file.
`-p` *prefix*	Uses the specified prefix for naming generated classes.
`-g`	Generates a *generic* version of the metadata that will access the MBeans through the MBean server instead of using a direct reference. This enables you to plug in dynamic MBeans, instead of the generated standard MBean skeletons.
`-gp` *prefix*	Uses the specified prefix to name the *generic* metadata classes. For example, the metadata class for group `System` will be named `System`*prefix*`Meta`. The default is no prefix.
`-sp` *prefix*	Uses the specified prefix to name the *standard* metadata classes. For example: the metadata class for group system will be named `System`*prefix*`Meta`. Default is no prefix.
`-help`	Prints a usage message explaining how to invoke the compiler, as follows: `<mib files>`: By default `mibgen` generates code only for the modules specified in the *first* file. The other files are only used for closure analysis except when the `-a` is specified.

The order followed by the `mibgen` compiler to find the `MIB_CORE` definitions file is as follows:

1. The user `MIB_CORE` definitions file specified in the MIB files using the `-mc mibgen` option.
2. The command line parameter specified using the `-Dmibcore.file` Java property.
3. The default `MIB_CORE` definitions file provided with the Java DMK in *installDir*`/etc/mibgen` (`mib_core.txt`). To succeed, you must be able to derive the installation directory from the `CLASSPATH` environment variable. Otherwise, the `mibgen` compiler looks for the `mib_core.txt` file in *currentDir*`/etc/mibgen`.
4. When using generic metadata (`-g` option), backward compatibility is not ensured. Using the `-g` option has generic advantages, whereby MBeans are accessed through the MBean server, any kind of MBeans can be plugged in, but this option slightly reduces the overall performance.

Note – SNMP MIB implementations generated using the `mibgen` compiler from Java DMK 5.0 can run and recompile on Java DMK 5.1 without modification.

1.3.1 Advanced `mibgen` Options

Advanced `mibgen` options are specified with a `-X` prefix, as shown in the following list.

`mibgen -X:`*advanced options mib files*

`-X:`*advanced options* includes the following options.

`-X:define`	Defines a valid `mibgen` property in the form *name=value*
`-X:use-display-hint[:`*on*`|:`*off*`]`	Uses `DISPLAY-HINT`. When `on`, this option instructs `mibgen` to generate an attribute of type `String` for any object using a textual convention whose `DISPLAY-HINT` is 255a [`com.sun.jdmk.tools.mibgen.options.use.display.hint=true`]
`-X:abstract-mib[:`*on*`|:`*off*`]`	Generate abstract MIB. When `on`, this option instructs `mibgen` to generate an abstract MIB. The MIB class is an abstract class when the MBean factory methods are abstract. [`com.sun.jdmk.tools.mibgen.options.mib.factory.abstract=true`]
`-X:no-table-access[:`*on*`|:`*off*`]`	No table accessor. When `on`, this option instructs `mibgen` not to generate any table accessors in the group of MBean interfaces. [`com.sun.jdmk.tools.mibgen.options.mbean.table.accessor=false`].
`-X:use-unsigned-long[:`*on*`|:`*off*`]`	Handles unsigned long values.
`-X:target:5.0`	Generates MIBs compatible with the Java DMK 5.0 implementation of SNMP.
`-X:help`	Print this help message.

1.4 Output From the `mibgen` Compiler

The `mibgen` compiler also generates the Java source code that is required for representing a whole MIB in an SNMP manager. The `mibgen` compiler parses an SNMP MIB and generates the following.

- For agents and managers:
  - A class mapping symbolic names with object identifiers of MIB variables
- For agents:
  - An MBean that represents the whole MIB
  - Classes that represents SNMP groups or entities as MBeans and their corresponding metadata classes
  - Classes that represents SNMP tables
  - Classes that represents SNMP enumerated types

MBeans generated by the `mibgen` compiler need to be updated to provide the definitive implementation. For more information, see the corresponding section in the *Java Dynamic Management Kit 5.1 Tutorial*.

1.4.1 Representation of the Whole MIB

The `mibgen` compiler generates a Java file that represents and initializes the whole MIB. This class extends the class `SnmpMib`. `SnmpMib` is an abstract Java class in the `com.sun.management.snmp.agent` package and is a logical abstraction of an SNMP MIB. The SNMP adaptor uses the `SnmpMib` class to implement agent behavior. The generated MIB file offers factory methods for group MBeans.

▼ To Implement the Generated MIB File

1. **Subclass the group MBean skeleton you want to implement, and complete the getter, checker, and setter methods.**
2. **Subclass the generated MIB file.**
3. **Redefine the factory methods for the group MBeans you have implemented, and ensure that they instantiate the actual implementation class and not the skeleton.**

The `mibgen` compiler uses the module name specified in the MIB definition to name files representing whole MIBs. The compiler removes special characters and replaces them with an underscore character (_).

1.4.2 Representation of the Whole MIB in an SNMP `OidTable`

The `mibgen` compiler generates a Java file that contains the code required to represent a whole MIB in an SNMP manager `OidTable`. This class extends the `com.sun.management.snmp.snmpOidTableSupport` class, which implements the `com.sun.management.snmp.snmpOidTable` class and maintains a database of MIB variables. A name can be resolved against the database. This file can be used by both the agent and the manager API and contains metadata definitions for the compiled MIB. The metadata can then be loaded into the `SnmpOid` table.

The file is always generated when `mibgen` is invoked. The generated file is called *MIBname*`OidTable`. The `-mo` option generates *only* the *MIBname*`OidTable` file. This file is the only file generated for SNMP managers. All other files are dedicated to the SNMP agents.

1.5 Classes Representing SNMP Groups

For each SNMP group defined in the MIB, the `mibgen` compiler generates:

- A skeletal MBean, with its interface
- A metadata file

1.5.1 Skeletal MBeans Representing Groups

The `mibgen` compiler generates an MBean for each group that is defined in the MIB. These skeletal MBeans need to be completed by adding implementation-specific code, to provide access methods. The generated code is initialized with default values for the various MIB variables. If the MIB specifies a default value for an SNMP variable, this value is used to initialize the corresponding variable in the MBean skeleton. Therefore, if you compile the generated code directly, you obtain a running agent. In this case, values returned by the agent when querying the MIBs will be default values or meaningless values, if no default value has been provided in the MIB file for the variable.

The `mibgen` compiler uses the group names specified in the MIB definition to name MBeans that are generated from groups.

1.5.2 Metadata Files

In addition to generating skeletal MBeans to represent each group, the `mibgen` compiler generates a metadata file. The metadata file contains Java source code that provides the SNMP view of the MBean. Metadata files do not need to be modified. For metadata files, the `Meta` suffix is added.

1.6 Classes Representing SNMP Tables

For each SNMP table defined in the MIB, the `mibgen` compiler generates:

- A class that contains the view of the table
- A metadata file that corresponds to the SNMP table
- A skeletal MBean that represents a table entry, with its interface
- A metadata file that corresponds to the skeletal MBean

1.6.1 Class Containing the SNMP View of a Table (Metadata Class)

The metadata class containing the SNMP view of a table contains all the management of the table index. Usually you do not need to subclass or access the generated table Metadata classes, except when implementing virtual tables. See SNMP Virtual Table example in the *Java Dynamic Management Kit 5.1 Tutorial* for details.

The class name is derived from the name of the table, and is postfixed by `Meta`. For example, for a table named `sysApplInstallPkgTable` in the MIB, `mibgen` will generate a `Metadata` class called `SysApplInstallPkgTableMeta`.

1.6.2 Class Containing the MBean View of the Table

The class containing the MBean view of a table enables you to add or remove entries dynamically from the table. This class contains callbacks and factory methods that enable you to instantiate or delete entries upon receiving requests from a remote SNMP manager. See the Simple SNMP Tables example and SNMP Table Insrtrumentation example in the *Java Dynamic Management Kit 5.1 Tutorial* for details.

The class name is prefixed with `Table`, followed by the name of the table. For example, for a table named `sysApplInstallPkgTable` in the MIB, `mibgen` will generate an MBean view of the table class called `TableSysApplInstallPkgTable`.

1.6.3 Skeletal MBeans Representing SNMP Table Entries

For each table in a MIB, the `mibgen` compiler generates an MBean that represents a table entry. These skeletal MBeans must be completed by adding implementation specific code, called access methods. The generated code is initialized with default values for table–entry fields. Therefore, if you compile the generated code directly, you obtain a running agent. In this case, values returned by the agent when querying the MIBs are not meaningful. The mibgen compiler uses the entry names that are specified in the MIB definition to name MBeans that are generated from table entries. For example, for a table entry definition named `sysApplInstallPkgEntry` in the MIB, `mibgen` will generate a skeletal MBean class named `SysApplInstallPkgEntry` and an interface named `SysApplInstallPkgEntryMBean`.

Note – Remote creation of table entries is disabled by default, for security reasons. You can dynamically enable and disable remote creation of table entries by calling the `setCreationEnabled` operation on the generated MBean-like object.

The `RowStatus` convention, that is defined in RFC 2579, is fully supported by the code generator. When a table is defined using SNMPv2, if it contains a control variable with row status syntax, the `mibgen` compiler generates a set of methods allowing this table to be remotely controlled by this variable. However, the remote creation and remote deletion of rows remains disabled by default.

Table objects are divided into two categories:

- A metadata class
- An MBean-like object

When remote table-entry creation is enabled, and the creation of a new table is requested, a factory method is called on the MBean-like object to instantiate the new table entry. By default, an instance of the skeleton class for that table entry is instantiated.

▼ To Instantiate Your Own Implementation Class

1. **Subclass the MBean-like object to redefine the factory method for remote entry creation.**
2. **Redefine this factory method so that it returns an instance of your implementation class, instead of the default skeleton.**
3. **Subclass this table's group MBean to instantiate your new MBean-like object, instead of the generated default object.**

This is demonstrated in the `RowStatus` example, which is presented in the *Java Dynamic Management Kit 5.1 Tutorial*.

1.6.4 Metadata Files

In addition to generating skeletal MBeans to represent each table entry, the `mibgen` compiler generates a Java file containing the SNMP view of the MBean. Metadata files do not need to be modified. For metadata files, the `Meta` suffix is added.

1.6.5 Classes Representing SNMP Enumerated Types

The `mibgen` compiler generates a specific class for each enumerated type that is defined in the MIB. This class contains all the possible values defined in the enumerated type. The generated class extends the generic class `Enumerated`, defined in the `com.sun.jdmk` package. The HTML adaptor can use the `Enumerated` class to display all the labels that are contained in an enumeration. The `mibgen` compiler can handle enumerated types defined as part of a type definition or in-line definition.

Generated code representing SNMP enumerated types is prefixed with `Enum` followed by the type name or the variable name for inline definition.

Note – The `mibgen` compiler has an option `-p` *prefix* that you can use to prefix the names of all generated files with a specific string.

For example, in MIB II, TCP connection states are represented by an enumeration containing all the possible states for a TCP connection. The `mibgen` compiler generates a Java class named `EnumTcpConnState` to represent the enumeration.

1.7 Information Mapping

For each group defined in your MIB, the `mibgen` compiler generates an MBean. Each variable in the group is represented as a property of the MBean. If the MIB allows read access to a variable, the `mibgen` compiler generates a getter method for the corresponding property. If the MIB allows write access to a variable, the `mibgen` compiler generates a setter method for the property. Tables are seen as indexed

properties whose type corresponds to the table entry type. The SNMP view of the table is maintained by a specific table object contained in the generated MBean. The `mibgen` compiler maps the MIB variable syntax to a well-defined Java type.

The MBeans that the `mibgen` compiler generates do not have any dependencies on specific SNMP objects. Therefore, these MBeans can be easily browsed or integrated into the various Java DMK components. The translation between the SNMP syntax and the MBean syntax is performed by the metadata.

MBeans generated by the `mibgen` compiler must be updated to provide the definitive implementation. The generated code is an operational agent. Thus, the code can be compiled, run, and tested without any modification.

As a general rule, use subclassing to implement your custom behavior. Do not edit the generated file, or your modification will be lost when you regenerate your MIB. Instead of subclassing the generated skeleton classes, you can also provide your own implementation that simply implements the corresponding generated interface, as shown in the SNMP Virtual Tables example in the *Java Dynamic Management Kit 5.1 Tutorial*.

Example 1–1 shows how to implement a skeletal MBean.

EXAMPLE 1–1 Implementing a Skeletal MBean

```
public class g1 implements g1MBean, Serializable {
        protected Integer myVar = new Integer (1);
      public g1(SnmpMib myMib) {
      {
      public Integer getMyVar() throws SnmpStatusException {
          return myVar;
      }

       public void setMyVar(Integer x)  throws SnmpStatusException {
           myVar = x;
      }

}
```

You must subclass or provide an implementation of the skeletal MBean to implement your MIB behavior. For information about developing an SNMP agent, SNMP Manager, SNMP API, and SNMP Proxy, see the corresponding sections in the *Java Dynamic Management Kit 5.1 Tutorial*.

CHAPTER 2

The HTML Protocol Adaptor

A protocol adaptor provides access to MBeans through a communications protocol. A protocol adaptor enables management applications to perform management operations on a Java Dynamic Management Kit (Java DMK) agent. For a Java DMK agent to be manageable, it must contain at least one adaptor. However, an agent can contain *many* adaptors, allowing it to be managed remotely through various protocols.

The HTML protocol adaptor acts as an HTML server. It enables web browsers to access agents through the HTTP communications protocol, to manage all MBeans in the agent. The HTML adaptor can be used as a tool for debugging and speeding the development of agents. The HTML protocol adaptor is implemented as a dynamic MBean.

The HTML protocol adaptor provides the following main HTML pages for managing MBeans in an agent:

- *Agent View*: Provides a list of object names of all the MBeans registered in the agent.
- *Agent Administration*: Registers and unregisters MBeans in the agent.
- *MBean View*: Reads and writes MBean attributes and performs operations on MBeans in the agent.

The HTML page displayed is generated by the HTML adaptor and enables you to perform the following operations on MBeans in the agen.:

- Read or write the attributes of an MBean instance
- Perform an operation on an MBean instance
- Instantiate an MBean
- Delete an MBean

2.1 HTML Connections

The HTML adaptor is an instance of the `com.sun.jdmk.comm.HtmlAdaptorServer` MBean. Your agent application must instantiate this class, register the MBean, and explicitly start the MBean by invoking its `start` method to allow HTML connections. When the HTML protocol adaptor is started, it creates a TCP/IP socket, listens for manager connections, and waits for incoming requests. By default, the HTML adaptor listens for incoming requests on port 8082. You can change this default value by specifying a port number:

- In the object constructor
- By using the `setPort` method before starting the adaptor

If a manager tries to connect, the `HtmlAdaptorServer` creates a thread which receives and processes all subsequent requests from this manager. The number of managers is limited by the `maxActiveClientCount` property. The default value of the `maxActiveClientCount` is 10.

When an `HtmlAdaptorServer` is stopped, all current connections are interrupted (some requests might be terminated abruptly), and the TCP/IP socket is closed. The `HtmlAdaptorServer` can perform user authentication. The `addUserAuthenticationInfo` method and the `removeUserAuthenticationInfo` method can be used to manage users and their corresponding authentication information. The HTML server uses the Basic Authentication Scheme, as defined in RFC 1945, section 11.1, to authenticate clients which connect to the server.

Before connecting a web browser to an agent, you must make sure that:

- The agent is running on a system that you can access by using the HTTP protocol
- The agent contains an instance of an HTML adaptor
- The compiled MBean classes are stored at a location that is specified in the `CLASSPATH` environment variable of the agent

To connect a browser to an agent, open the page given by the following URL in a web browser.

`http://`*`host`*`:`*`port`*

In the URL above, *host* is the host name of the machine on which the agent is running. The *port* is the port number used by the HTML adaptor in the agent. The default port number is 8082.

2.2 Limitations of the HTML Protocol Adaptor

The HTML protocol adaptor has the following limitations:

- The minimum value for the reload period is 5 seconds. The value 0 defaults to no reloading.
- Arrays of classes are always displayed in read-only mode.
- Arrays of dimension 2 and higher are not fully expanded.
- Supported attribute types for reading and writing are as follows.

  - `boolean boolean[] Boolean Boolean[]`
  - `byte Byte Byte[]`
  - `char char[] Character Character[]`
  - `Date Date[]` (for example, July 21st, 2002 8:49:04 PM CEST)
  - `double double[] Double Double[]`
  - `float float[] Float Float[]`
  - `int int[] Integer Integer[]`
  - `long Long Long[]`
  - `Number`
  - `javax.management.ObjectName javax.management.ObjectName[]`
  - `short Short Short[]`
  - `String String[]`

  In addition, `com.sun.jdmk.Enumerated` is supported for readable attributes. Because `com.sun.jdmk.Enumerated` is an abstract class, only write-only attributes whose actual subclass is declared in the signature of its setter can be set through the HTML adaptor.

Note – For unsupported read-only attribute types, if not null, the `toString()` method is called. If the getter of a read-only or a read-write attribute throws an exception, the thrown exception name and message are displayed. In this case, this attribute cannot be set through the HTML adaptor even if it is a read-write attribute.

- Supported operation and constructor parameter types are as follows:

  - `boolean Boolean`
  - `byte Byte`
  - `char Character`
  - `Date` (for example, July 21st, 2002 8:49:04 PM CEST)
  - `double Double`
  - `float Float`

- `int Integer`
- `long Long`
- `Number`
- `javax.management.ObjectName`
- `short Short`
- `String`

Note – When reading a value of type `Number`, the server tries to convert it first to an `Integer`, then a `Long`, then a `Float`, and finally a `Double`. The server stops at the first primitive type that succeeds.

Use the "Reload" button displayed in the HTML page of an MBean view rather than the reload button of the web browser. Otherwise, you might reinvoke the setters of all attributes, if this was your last action.

CHAPTER 3

Legacy MBean Proxy Generator (`proxygen`)

You can use the `proxygen` tool supplied with the Java Dynamic Management Kit (Java DMK) to generate a proxy MBean from its corresponding MBean. A proxy MBean is an image of an agent-side MBean that exists on the manager side. The `proxygen` tool allows you to customize your proxy MBeans depending on how you want to use them in your management application. For more information about the relationship between MBeans and proxy MBeans, see the *Java Dynamic Management Kit 5.1 Getting Started Guide*.

Note – The `proxygen` tool is marked as "deprecated" in Java DMK 5.1. Use the `proxygen` tool only if you require proxies for legacy remote method invocation (RMI), hypertext transfer protocol (HTTP), and secure HTTP (HTTP/S) connectors. For new RMI, RMI/IIOP and Java Management Extensions messaging protocol (JMXMP) connectors that comply with the JMX 1.2 and JMX Remote API 1.0 specifications, you can generate a proxy object at runtime, given just its Java interface. These dynamic proxies cannot be used with the legacy connectors. For more information on dynamic proxies, see the *Java Dynamic Management Kit 5.1 Tutorial*.

This chapter describes how to use the `proxygen` tool, in the following sections:

3.1 Overview of the proxygen Tool

The proxygen tool takes the compiled Java class of an MBean and generates the Java interface and Java proxies. The Java proxies consist of Java source code that implements the interface. To develop a Java manager with code generated by proxygen, you call the methods of the proxy MBean's interface.

Options of the proxygen tool enable you to modify the characteristics of the proxies you generate from an MBean. For example, with some options, you can generate read-only or read-write proxies. By generating from the same MBean a set of proxies with different characteristics, you can develop a Java manager whose behavior is modified at runtime, depending on which proxies are loaded. For example, when the read-only proxies are loaded, the Java manager cannot modify properties in the MBean.

A proxy MBean consists of two components:

- A Java interface that defines which operations of the MBean are accessible to a Java manager
- A Java class that implements the operations defined in the Java interface

For example, if you have an MBean *MyClass*, the proxygen tool gives you a proxy MBean that consists of the following files.

- *MyClassProxyMBean*.java: the Java interface
- *MyClassProxy*.java: the Java class

The proxygen tool generates Java source code, not compiled Java classes. For your proxy MBeans to be accessible to a Java manager, you must compile the files that proxygen generates. Then you must make sure that the compiled Java classes are stored at the location specified by the CLASSPATH environment variable of the manager, or are accessible through the class loader of the manager.

3.2 Starting the proxygen Tool

To start the proxygen tool, type the command for your operating environment:

```
prompt% installDir/SUNWjdmk/5.1/bin/proxygen <options>  <classes>
```

Alternatively, invoke the java com.sun.jdmk.tools.ProxyGen class by first invoking java com.sun.jdmk.tools.ProxyGen <options> <classes>. Provide the class name without the .class extension.

Note – The script for starting the `proxygen` tool uses the `JAVA_HOME` environment variable to determine the path to the Java 2 Platform Standard Edition (J2SE). Therefore, even if you have the correct path to the J2SE platform in your `PATH` environment variable, this setting is overwritten by the `JAVA_HOME` variable.

3.3 `proxygen` Tool Options

The `proxygen` command takes options, as follows:

`proxygen` *options classes*

The *options* include the following.

`-d` *dir*	Specifies a destination directory for the generated code.
`-ro`	Generates read-only proxy MBeans. Calling setter methods of these read-only proxies raises a `com.sun.jdmk.RuntimeProxyException`.
`-tp` *pkgName*	Generates code in the target package specified by *pkgName*.
`-classpath` *path*	Specifies a class path to use for locating the class to compile. By default, the system class path is used.
`-help`	Prints a message that briefly describes each `proxygen` option.

3.4 Example of the `proxygen` Tool

The following example shows how to generate the managed object for the `Simple` class and `SimpleMBean` interface. You must compile the classes before using the `proxygen` tool to generate the managed object. Finally, you must compile the Java code generated by the `proxygen` tool.

The source code for the `Simple` class is contained in the *installDir*`/SUNWjdmk/5.1/examples/legacy/MonitorMBean` directory, where *installDir* is the directory under which the Java DMK was installed.

Note – On the Solaris platform, the *installDir* file hierarchy is not writable by default. In this case you must copy the `Simple.java` and `SimpleMBean.java` files to a directory where you have write permissions.

▼ To Generate the Managed Object for the `Simple` Class

1. **Add the following to your `CLASSPATH`:**

   - In the Solaris or Linux operating environment.

     installDir`/SUNWjdmk/5.1/lib/jdmkrt.jar`
     installDir`/SUNWjdmk/5.1/lib/jdmktk.jar`
     installDir`/SUNWjdmk/5.1/lib/jmx.jar`

   - In a Windows 2000 operating environment.

     installDir`\SUNWjdmk\5.1\lib\jdmkrt.jar`
     installDir`\SUNWjdmk\5.1\lib\jdmktk.jar`
     installDir`\SUNWjdmk\5.1\lib\jmx.jar`

2. **Type the following commands.**

```
prompt% javac Simple.java SimpleMBean.java
prompt% /.../bin/proxygen -classpath . Simple
Destination directory set to ./.

Starting compilation of Simple.

Starting to generate stub SimpleProxy.java for class Simple
Starting to generate MBean interface SimpleProxyMBean.java for class Simple
```

   Proxy MBeans generated using the `proxygen` tool in Java DMK 4.2 must be regenerated to run in Java DMK 5.1, because some of the methods used in version 4.2 have been deprecated. However, proxy MBeans generated using the `proxygen` tool in Java DMK 5.0 do not need to be regenerated to run in Java DMK 5.1.

3.5 Output of the `proxygen` Tool

For an MBean defined in the Java class *BeanName*, the `proxygen` tool generates the following.

- A Java interface (*BeanName*`ProxyMBean`), which defines the methods of the MBean that are accessible to a Java manager

- A Java proxy (*BeanName*`Proxy`), which implements the methods defined in the Java interface *BeanName*`ProxyMBean`

For example, when an MBean representing a Java class named `Simple` is compiled, the `proxygen` tool generates the source code for the following classes.

- A Java interface named `SimpleProxyMBean`
- A Java class named `SimpleProxy`, which implements the `SimpleProxyMBean` interface

3.6 Mapping Rules

The `proxygen` tool uses the Java Reflection API for analyzing an MBean and generating its associated proxy MBean. The `proxygen` tool parses an MBean using the JMX-specific design patterns. The mapping rules that `proxygen` uses for generating the proxy MBean are described in the following subsections.

3.6.1 Mapping of Attributes

The `proxygen` tool generates code only for exposed operations of the MBean itself. Each attribute of the MBean is present in the proxy MBean with the same accessor getter and setter methods. Therefore, if an attribute is read-only in the MBean, the property is read-only in the generated proxy MBean.

3.6.2 Mapping of Operations

In addition to the attribute accessors, the `proxygen` tool generates code only for exposed operations of the MBean itself.

3.6.3 Methods in the `Proxy` Interface

The proxy MBeans that the `proxygen` tool generates also contain methods that are not present in the MBean. These methods are defined in the Java interface `com.sun.jdmk.Proxy`. The proxy MBean that is generated implements this interface. These methods are public methods that do not follow the design patterns defined by the JavaBeans component model.

These methods provide additional functionality for proxy MBeans and the management applications that instantiate them. The purpose of these methods is twofold:

- To make sure that the information provided by the proxy MBean is up–to–date. For example, methods are defined for binding and unbinding a proxy MBean from a remote MBean server.
- To get the object name and class of the remote MBean represented by the proxy MBean.

3.7 Using the Generated Code

The `proxygen` tool generates Java source code that you can use for developing Java managers. To develop a Java manager with code generated by the `proxygen` tool, use the `RemoteMBeanServer` interface. By using this interface, you can develop Java managers without having to modify the code that the `proxygen` tool generates.

Nevertheless, if you want to define a specific view of an MBean, you can modify the code that the `proxygen` tool generates. To ensure that the modified code remains consistent with the MBean that it represents, modify only the proxy and not the interface.

CHAPTER 4

Tracing Mechanism

This chapter explains how to use the tracing mechanism to help you trace or debug the Java Dynamic Management Kit (Java DMK) API. The tracing mechanism gives you internal runtime information, and you can specify the information type and level of trace and debug information you want to receive.

Note – The proprietary tracing mechanism classes described in this chapter are all deprecated as of Java DMK 5.1. These classes have all been superseded by the `java.util.logging` classes that are now supported by the Java Management Extensions (JMX). How to activate logging using the `java.util.logging` API is explained in "4.1 Activating the `java.util.logging` API" on page 31. The information contained in the rest of this chapter is deprecated and is retained for purposes of backwards compatibility only.

4.1 Activating the `java.util.logging` API

To activate traces using the standard `java.util.logging` API, you must edit, or copy and edit, the `template.logging.properties` file that is provided by Java DMK in the directory *installDir*`/etc/conf/`.

You must uncomment the categories from which you want to obtain traces, and the level of tracing you require. Then pass the file on the command line when starting your Java classes, as follows

```
$ java -Djava.util.logging.config.file=Path_to_logging.properties_file  Java_class
```

4.2 Receiving Trace and Debug Information Using the Deprecated `TraceManager`

To receive trace and debug information you must add a notification listener to the class `com.sun.jdmk.TraceManager`.

You control the tracing by defining the trace properties specific to the Java DMK. Three factors affect tracing:

- Various components that send trace messages
- Level of detail
- Output destination

The `com.sun.jdmk.trace.Trace` class is used to emit trace messages. All the classes of the Java DMK use this `Trace` class for sending traces. You can use the `Trace` class in your own code for producing debug traces for your own classes.

The `com.sun.jdmk.TraceManager` class provides methods for receiving trace and debug messages. Options provided by the `TraceManager` class are described in the following sections:

- "4.2 Receiving Trace and Debug Information Using the Deprecated `TraceManager`" on page 32
- "4.3 Specifying the Type of Trace and Debug Information" on page 33
- "4.4 Specifying the Level of Trace and Debug Information" on page 34

The `com.sun.jdmk.TraceManager` class uses the notification mechanism to distribute the information. You must add a notification listener to receive information (see example Example 4–1). There are two ways to receive trace information.

- Adding a notification listener with a filter in the code. It is possible to have more than one notification listener but with different filters. With `TraceFilter`, you can specify the type and level of information you want to receive. See Example 4–2 and Example 4–3.
- Specifying system properties in the command to start the Java interpreter when you run a class. In this case, the code of the class must include a call to the `TraceManager` method. When the `TraceManager` method is called, all the previously enabled trace and debug information are disabled. Only the properties currently defined when the method is called are enabled.

EXAMPLE 4–1 Creating a Notification Listener

```
// Create a listener and save all info to the file /tmp/trace
TraceListener listener = new TraceListener("/tmp/trace");
```

EXAMPLE 4–2 Creating a Trace Filter

```
// create a trace filter with LEVEL_DEBUG and INFO_ALL/
TraceFilter filter = new TraceFilter(Trace.LEVEL_DEBUG, Trace.INFO_ALL);
```

EXAMPLE 4–3 Adding the Notification Listener to the class

```
// add the listener to the class Trace/
TraceManager.addNotificationListener(listener, filter, null);
```

4.3 Specifying the Type of Trace and Debug Information

It is possible to specify the type of trace and debug information you want to receive. The following types are specified.

- `INFO_MBEANSERVER`: information about the MBean server
- `INFO_MLET`: information from an m-let service
- `INFO_MONITOR`: information from a monitor
- `INFO_TIMER`: information from a timer
- `INFO_ADAPTOR_CONNECTOR`: information concerning all adaptors and connectors
- `INFO_ADAPTOR_HTML`: information from an HTML adaptor
- `INFO_CONNECTOR_RMI`: information from a RMI connector
- `INFO_CONNECTOR_HTTP`: information from HTTP connectors
- `INFO_CONNECTOR_HTTPS`: information from HTTPS connectors
- `INFO_ADAPTOR_SNMP`: information from an SNMP adaptor
- `INFO_DISCOVERY`: information from a discovery service
- `INFO_SNMP`: information from an SNMP manager service
- `INFO_NOTIFICATION`: information from notification mechanism
- `INFO_HEARTBEAT`: information from heartbeat mechanism
- `INFO_RELATION`: information from relation service
- `INFO_MODELMBEAN`: information from the model MBean components
- `INFO_MISC`: information sent from any other classes
- `INFO_ALL`: information from all classes

The preceding information is held by the `TraceTags` class

4.4 Specifying the Level of Trace and Debug Information

The level of detail controls the number of messages you receive. The *trace* level is the default that gives information about the actions of the MBean server and other components. The *debug* level includes all the trace information providing information to help diagnose Java DMK implementation. If this level is specified, the information of `LEVEL_TRACE` is sent too. It is possible to specify the level of trace or debug information you want to receive. Two levels of information are specified in the `TraceTags` class.

`LEVEL_TRACE`	Provides information to help a developer when programming
`LEVEL_DEBUG`	Provides information to help diagnose Java DMK implementation

If you choose the second option, you automatically receive all trace information as well as debug information. By default, the level is set to `LEVEL_TRACE`.

Index

S

T

W